NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DECOMPOSITION RECOVERY EXTENSION TO THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS) CHANGE-MERGE TOOL

by

William Ronald Keesling

September 1997

Thesis Advisor:	Valdis Berzins
Co-Advisor:	Luqi

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1997	Master's Thesis

4. TITLE AND SUBTITLE

DECOMPOSITION RECOVERY EXTENSION TO THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS) CHANGE-MERGE TOOL

5. FUNDING NUMBERS

6. AUTHOR

Keesling, William Ronald

7. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSOR/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense of the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A promising use of Computer Aided Prototyping System (CAPS) is to support concurrent design. Key to success in this context is the ability to automatically and reliably combine and integrate the prototypes produced in concurrent efforts. Thus, to be of practical use in this as well as most prototyping contexts, a CAPS tool must have a fast, automated, reliable prototype integration capability.

The current CAPS Change-Merge Tool is fast, automated, and uses a highly reliable formalized *semantics-based change-merging method* to integrate, or *change-merge*, prototypes which are written in Prototype System Description Language (PSDL). This method can guarantee correct merges, but it loses the prototype's design decomposition structure in the process. The post-merge prototype is fully functional, but the design decomposition structure vital to prototype understandability must be manually recovered before post-merge prototyping can continue. The delay incurred is unacceptable in a rapid prototyping context.

This thesis presents a software design and Ada implementation for a formalized algorithm which extends the current CAPS Change-Merge Tool to automatically and reliably recover a merged prototype's design decomposition structure. The algorithm is based in formal theoretical approaches to software change-merging and includes a method to automatically report and resolve structural merge conflicts. With this extension to the Change-Merge Tool, CAPS prototyping efforts, concurrent or otherwise, can continue post-merge with little or no delay.

14. SUBJECT TERMS	15. NUMBER OF PAGES
CAPS, Software Change-Merging, Automated Software Integration, Software Prototyping, Concurrent Prototyping	204
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	**UNCLASSIFIED**	**UNCLASSIFIED**	**UL**

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std 239-18

Approved for public release; distribution is unlimited

DECOMPOSITION RECOVERY EXTENSION TO THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS) CHANGE-MERGE TOOL

William Ronald Keesling
B.S., San Diego State University, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1997

Author: William Ronald Keesling
William Ronald Keesling

Approved by: Valdis Berzins
Valdis Berzins, Thesis Advisor

Luqi, Co-Advisor

Timothy J. Shimeall for Ted Lewis
Ted Lewis, Chairman, Department of Computer Science

ABSTRACT

A promising use of Computer Aided Prototyping System (CAPS) is to support concurrent design. Key to success in this context is the ability to automatically and reliably combine and integrate the prototypes produced in concurrent efforts. Thus, to be of practical use in this as well as most prototyping contexts, a CAPS tool must have a fast, automated, reliable prototype integration capability.

The current CAPS Change-Merge Tool is fast, automated, and uses a highly reliable formalized *semantics-based change-merging method* to integrate, or *change-merge*, prototypes which are written in Prototype System Description Language (PSDL). This method can guarantee correct merges, but it loses the prototype's design decomposition structure in the process. The post-merge prototype is fully functional, but the design decomposition structure vital to prototype understandability must be manually recovered before post-merge prototyping can continue. The delay incurred is unacceptable in a rapid prototyping context.

This thesis presents a software design and Ada implementation for a formalized algorithm which extends the current CAPS Change-Merge Tool to automatically and reliably recover a merged prototype's design decomposition structure. The algorithm is based in formal theoretical approaches to software change-merging and includes a method to automatically report and resolve structural merge conflicts. With this extension to the Change-Merge Tool, CAPS prototyping efforts, concurrent or otherwise, can continue post-merge with little or no delay.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I would like to thank Dr. Berzins for suggesting that I take on this interesting thesis topic and for his help along the way. I would also like to thank Mr. Duston Hayward, Dr. Shing, and all involved at NRaD and Naval Postgraduate School for making the distance learning program possible, and for giving me the opportunity to pursue a Masters Degree through Naval Postgraduate School.

I. INTRODUCTION

This thesis presents a software design and Ada implementation for the **decompose_graph** algorithm presented in [Ref. 1]. The **decompose_graph** algorithm extends the capability of the Computer Aided Prototyping System's (CAPS) Change-Merge Tool to automatically combine and merge changes to a prototype's design decomposition structure [Ref. 1, 2].

The purpose of CAPS is to facilitate rapid prototyping of hard real-time systems, especially systems of medium to large size [Ref. 2]. A promising use of CAPS is in support of concurrent design efforts where separate design teams prototype different aspects of a system in parallel. These parallel efforts can significantly reduce system design time and result in significant cost savings.

Key to success of concurrent prototyping is the ability to automatically combine and integrate the prototypes produced in concurrent efforts. Thus, to be of practical use in this, as well as most, rapid prototyping contexts, CAPS must have a fast, automated, reliable integration capability.

The current CAPS integration capability, the Change-Merge Tool, is fast, automated, and uses a highly reliable formalized *semantics-based change-merging method* to integrate, or *change-merge*, prototypes which are written in Prototype System Description Language (PSDL) [Ref. 2, 3]. This method, based on *prototyping slicing*, guarantees correct results for conflict-free merges [Ref. 2, 4]. However, the use of prototype slicing requires the decomposition structure of the prototypes to be removed prior to merge. The merge results in a fully functional prototype, but the prototype's decomposition structure is lost in the process.

For PSDL prototypes of any size or complexity, decomposition structure is vital to prototype understandability, and thus critical to success of sustained rapid prototyping. In the case of the current CAPS Change-Merge Tool, the design decomposition structure lost in change-merge must be *manually* recovered before post-merge prototyping can continue. The delay incurred is unacceptable – it effectively takes the *rapid* out of *rapid prototyping*. [Ref. 1]

The **decompose_graph** algorithm extends the CAPS Change-Merge Tool to automatically recover the design decomposition structure lost in the prototype change-merge and reconstruct a prototype which has a decomposition structure which accurately reflects structural changes. The algorithm is based in formal theoretical approaches to software change-merging which "... work on special kinds of lattices that are also Brouwerian or Boolean algebras…." [Ref. 1], and includes a method to automatically report and resolve structural merge conflicts. With this extension, post-merge prototyping with CAPS can continue without the unacceptable delay incurred by the need to manually recover design decomposition structure.

In the context of this thesis, the **decompose_graph** algorithm is applied to PSDL prototypes. However, the algorithm also has applicability "…to the informal dataflow diagrams commonly used in requirements modeling and software design…". [Ref. 1]

The purpose of the following chapters and appendices is to provide the reader with an understanding of the software design and Ada implementation of the **decompose_graph** algorithm. The formal theory underlying **decompose_graph** given in [Ref. 1] is not restated.

Chapter II of this thesis provides an overview of PSDL prototype structural decomposition in the context of decomposition recovery. Chapter III gives an overview of what can be considered the two stages of PSDL prototype decomposition structure recovery. Chapter IV presents a detailed design for **decompose_graph**. Chapter V

provides an introduction to **decompose_graph** implementation, discusses test results, and directs the reader to Appendix A for source code listings. Chapter VI presents conclusions. Appendix B lists changes to PSDL_TYPE abstract data type necessary to accommodate the implementation of **decompose_graph**. Appendix C lists test-cases, test-drivers, and test results.

II. PSDL PROTOTYPES AND DECOMPOSITION RECOVERY

PSDL is a high level specification and design language developed for use in the CAPS development environment for specifying and designing prototype software systems [Ref. 1]. It is detailed extensively in [Ref. 3] and [Ref. 4]. In brief,

> PSDL represents software systems as generalized dataflow diagrams annotated with timing and control constraints...The notation is executable and has a formal semantics...that is a compatible refinement of informal dataflow diagrams traditionally used in software design. A PSDL prototype is a hierarchical network of components. [Ref. 1]

All PSDL operators have a specification and implementation part. The implementation part of an atomic operator specifies an executable module written in a language such as Ada or C. The implementation part of a composite operator is a graph which contains atomic or composite operators as vertices, the data streams which connect the operators as edges, and sets of timing and control constraints which restrict the behavior of these operators and data streams. [Ref. 1]

Every PSDL prototype has at least has one composite operator – the root operator. In Figure 2.1, this is the doubled circled operator labeled "ROOT". (In Figures 2.1 through 2.11, composite operators are indicated by double circles and atomic operators by single circles.) Composite operators represent a grouping of operators based on some design criteria (e.g., commonality of function). They are the feature of PSDL that facilitates top-down design decomposition for prototypes. They represent a point, or level, of design decomposition.

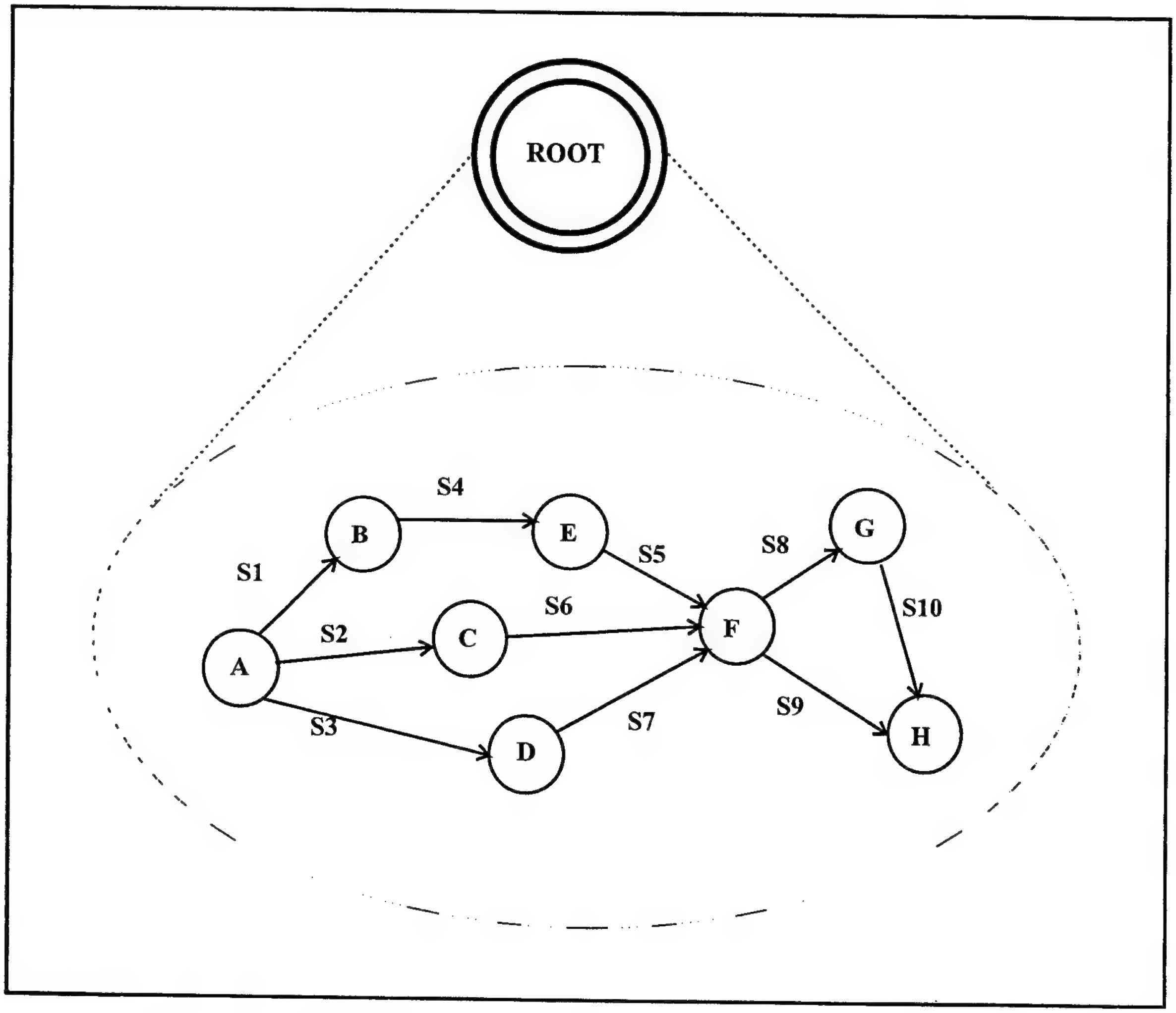

Figure 2.1: PSDL Prototype with One Composite Operator – ROOT

In terms of the functionality, composite operators are virtual – the functionality of the composite resides in its child operators. For example, assuming that the atomic operators in Figures 2.1 and 2.2 are functionally equivalent, the prototype in Figure 2.2 is functionally equivalent to the prototype in Figure 2.1. The functionality of composite operator CO1 resides in atomic operators C and D, and the functionality for composite operator CO2 resides in atomic operators G and H. Figure 2.3 gives an actual PSDL specification for a composite operator named **gui_in** and one of its child atomic operators, **gui_input_event_monitor**.

For the purpose of understanding the **decompose_graph** algorithm and decomposition recovery, there are several ways to view a PSDL prototype. One view is as suggested above: as a hierarchy of directed data-flow graphs where the vertices are operators and the edges are data streams. The implementation graphs of composite operators capture this view. Figure 2.1 gives a simple example of the implementation graph for the root composite operator of a simple prototype which has all atomic operators. The operators are labeled A through H and the data steams are labeled S1 through S10.

Figure 2.1 is also an example of a post-merge flattened (expanded) prototype. It has only one composite operator – the root operator, and thus only one implementation graph. The parent of every atomic operator in a flattened prototype is the root operator. A prototype which has decomposition structure would have the same number of atomic operators as its flattened version, but many of these operators would be allocated to the graphs of other composite operators.

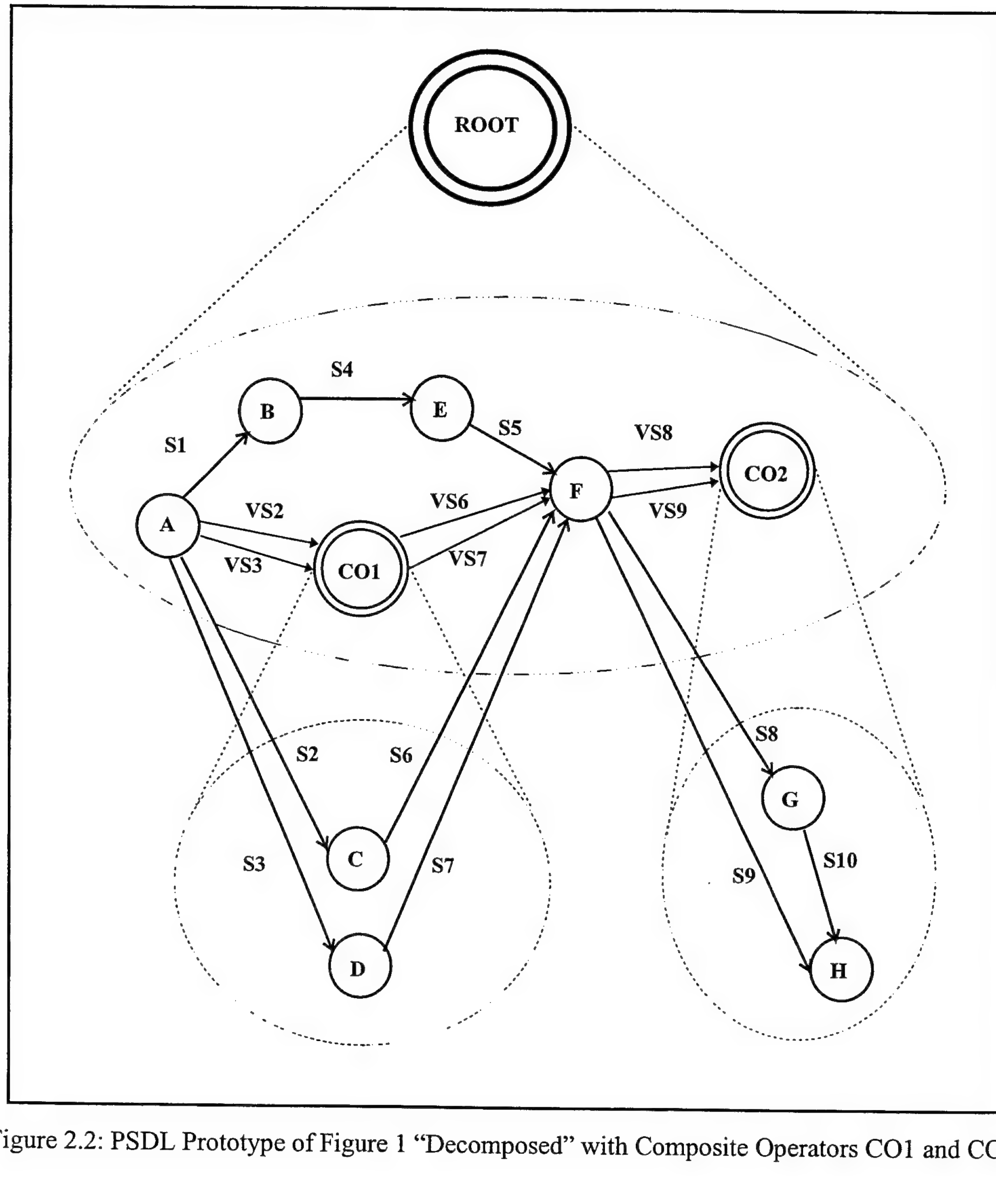

Figure 2.2: PSDL Prototype of Figure 1 "Decomposed" with Composite Operators CO1 and CO2

```
OPERATOR gui_in
 SPECIFICATION
  OUTPUT
   gui_in_str : my_unit
END
IMPLEMENTATION
 GRAPH
  VERTEX choose_inputs : 200 MS
  VERTEX gui_input_event_monitor : 200 MS

  EDGE gui_in_str
   choose_inputs -> EXTERNAL

 CONTROL CONSTRAINTS
  OPERATOR choose_inputs
   PERIOD 2000 MS

  OPERATOR gui_input_event_monitor
END

OPERATOR gui_input_event_monitor
 SPECIFICATION
  MAXIMUM EXECUTION TIME 200 MS
END
IMPLEMENTATION ADA gui_input_event_monitor
END
```

Figure 2.3: Example PSDL Specification for a Composite Operator and an Atomic Operator

Figure 2.2 illustrates this idea where the flat prototype of Figure 2.1 has been *decomposed* with the addition of composite operators CO1 and CO2. As Figure 2.2 attempts to illustrate, atomic operators C and D are now in the graph of CO1, and atomic operators H and G are now in the graph of CO2. The dashed circles and lines indicate operators and data streams in a composite operator's graph. Also note the labels for the data streams directed to and from CO1 and CO2. They are prefixed with a V to indicate their virtual nature – actual data streams are directed to and from atomic operators only. Data streams to and from composite operators are for the most part understandability aids for graphical display of prototypes in CAPS display tools. Figures 2.4 through 2.6 illustrate this idea. Figure 2.4 gives a root level view of the prototype of Figure 2.2, and Figures 2.5 and 2.6 give a view at the CO1 and CO2 decomposition level respectively.

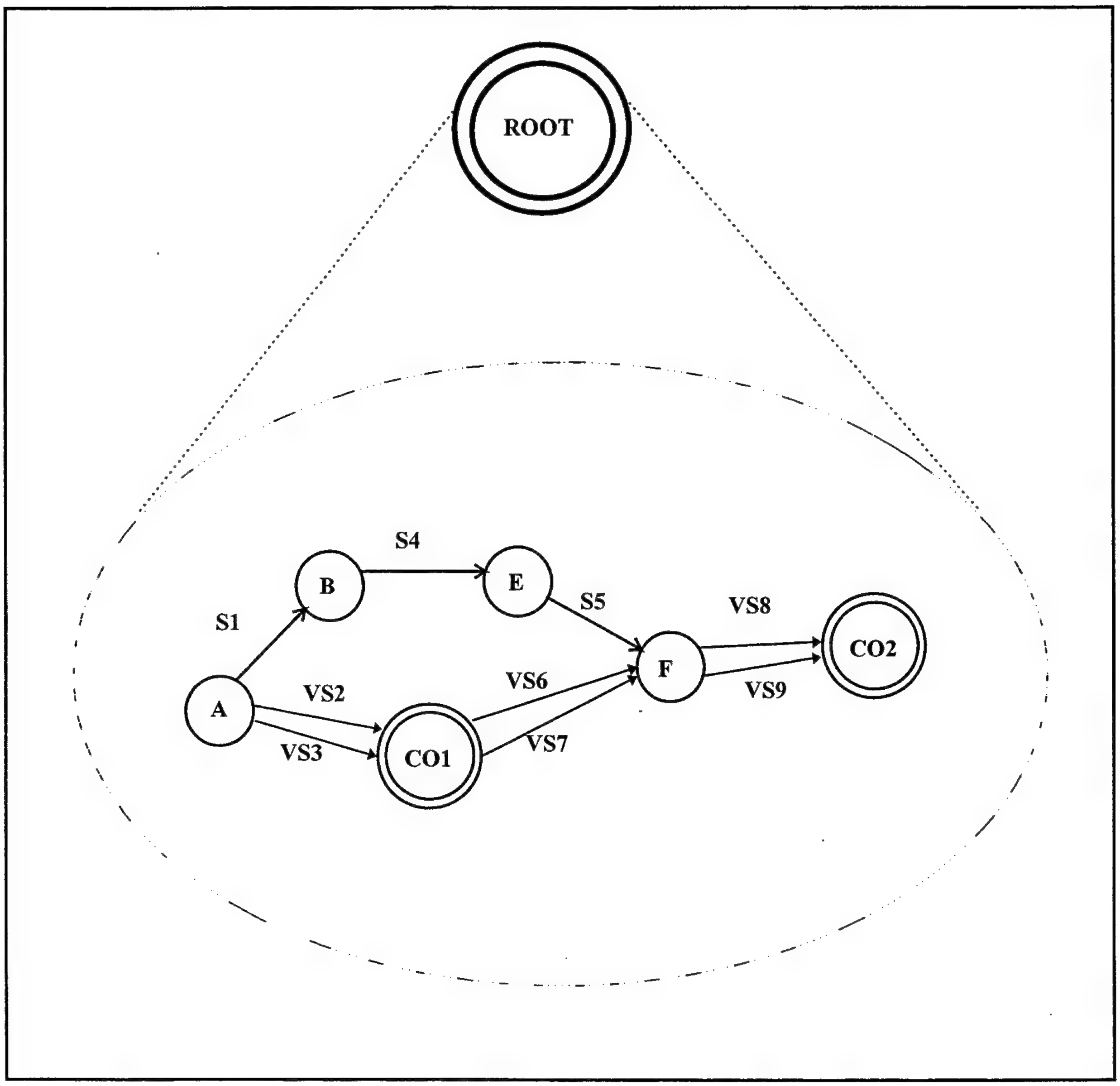

Figure 2.4: ROOT Operator Decomposition view for PSDL Prototype of Figure 2.2.

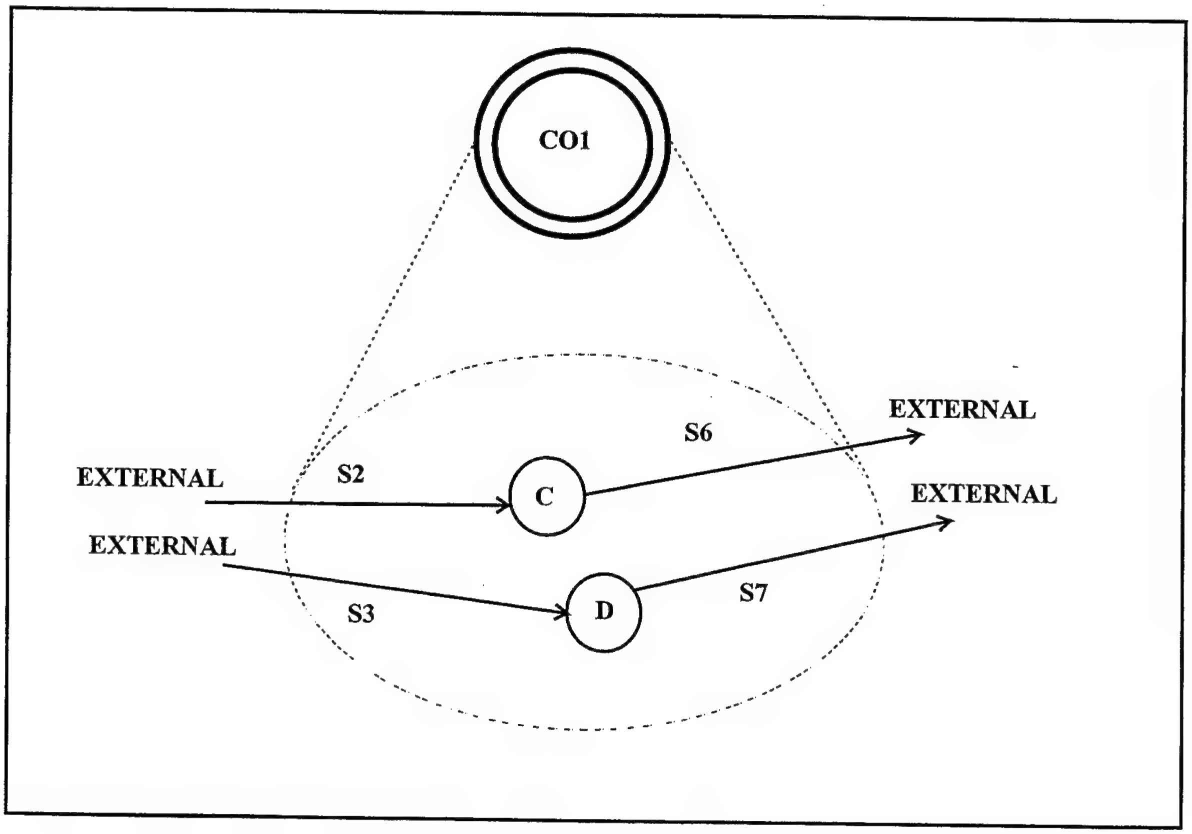

Figure 2.5: CO1 Operator Decomposition view for PSDL Prototype of Figure 2.2.

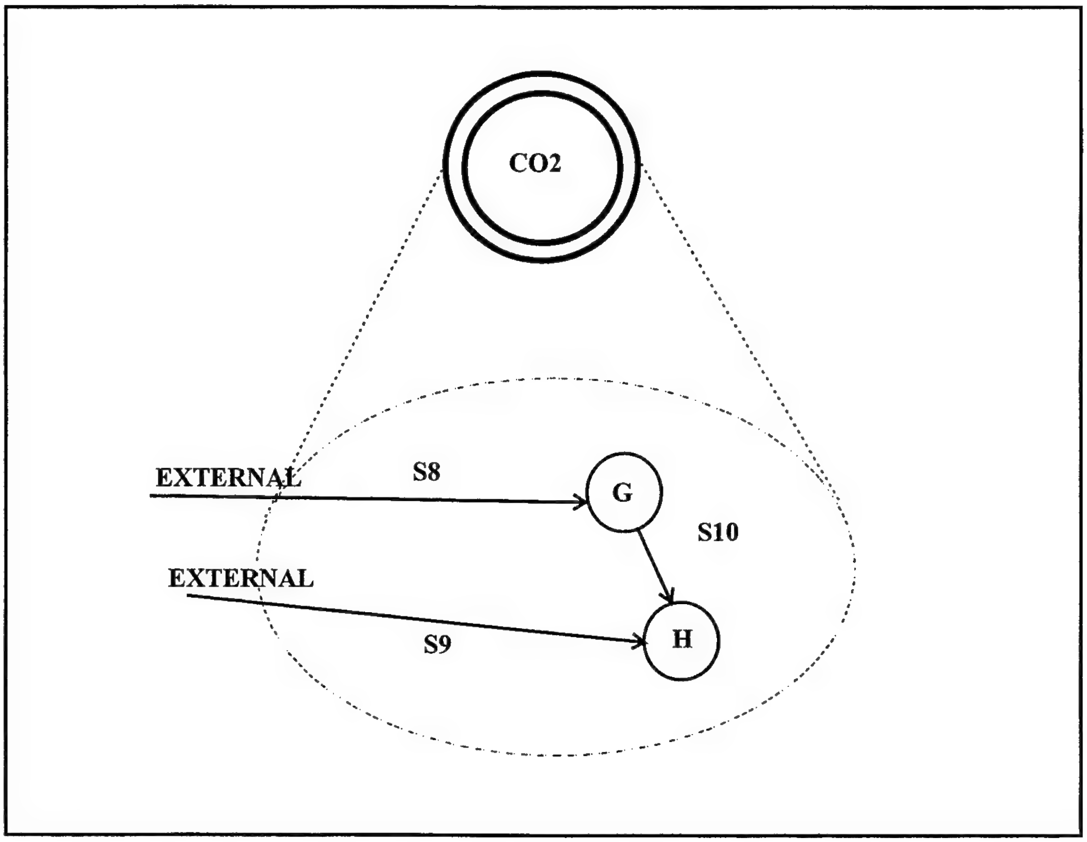

Figure 2.6: CO1 Operator Decomposition view for PSDL Prototype of Figure 2.2.

Another view of a PSDL prototype is as a graph where nodes are operators and the edges are parent-child relationships. Leaf nodes in this view are atomic operators and all other nodes are composite operators. The basic decomposition structure for a prototype is captured in this parent-child relationship. Figures 2.7 and 2.8 illustrate this simplified view. Figure 2.7 represents the parent-child relationship view for the flat prototype of Figure 2.1, and Figure 2.8 represents the parent-child relationship view for the prototype of Figure 2.2. In Figure 2.8, operator D is a child of CO1, and operator CO1 is a child of ROOT. (In passing, note that the ordered sequence <ROOT, CO1> is the *ancestor chain* for operator D.)

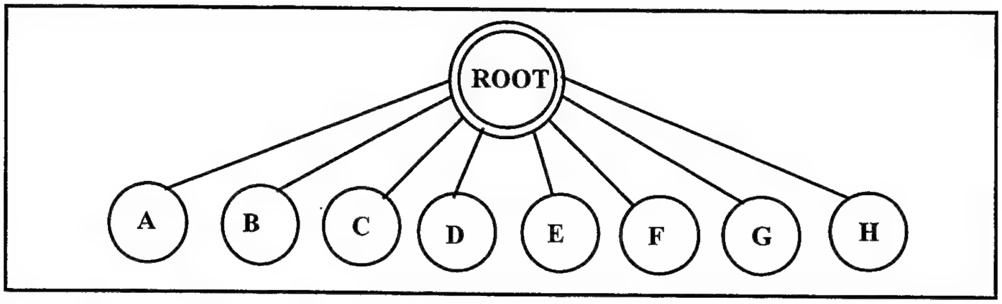

Figure 2.7: Parent-Child Relationship Graph for PSDL Prototype of Figure 2.1.

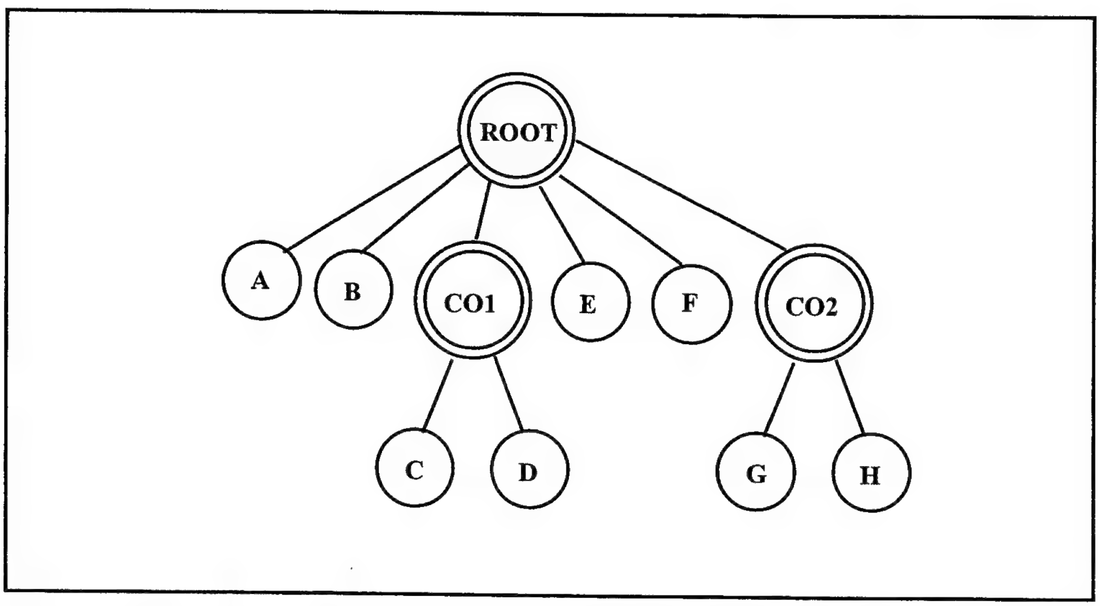

Figure 2.8: Parent-Child Relationship Graph for PSDL Prototype of Figure 2.2.

To illustrate what **decompose_graph** does, assume a BASE version of a PSDL prototype a given in Figure 2.9. CO1 of Figure 2.5 is added to the BASE version creating CHANGE A as illustrated in Figure 2.10. CO2 of Figure 2.6 is added to the BASE version creating CHANGE B as illustrated in Figure 2.11. CHANGE A and CHANGE B are now merged with the BASE to create a new version of the prototype. The desired result of the merge is the prototype of Figure 2.2. However, the current CAPS Change-Merge Tool will produce the flat prototype of Figure 2.1. As mentioned previously, the flat prototype is fully functional, but the design captured in the BASE, CHANGE A and CHANGE B decomposition structures is lost. The **decompose_graph** algorithm extends the CAPS merge tool to produce the *decomposed* merged prototype of Figure 2.2 instead of the *flattened* merged prototype of Figure 2.1.

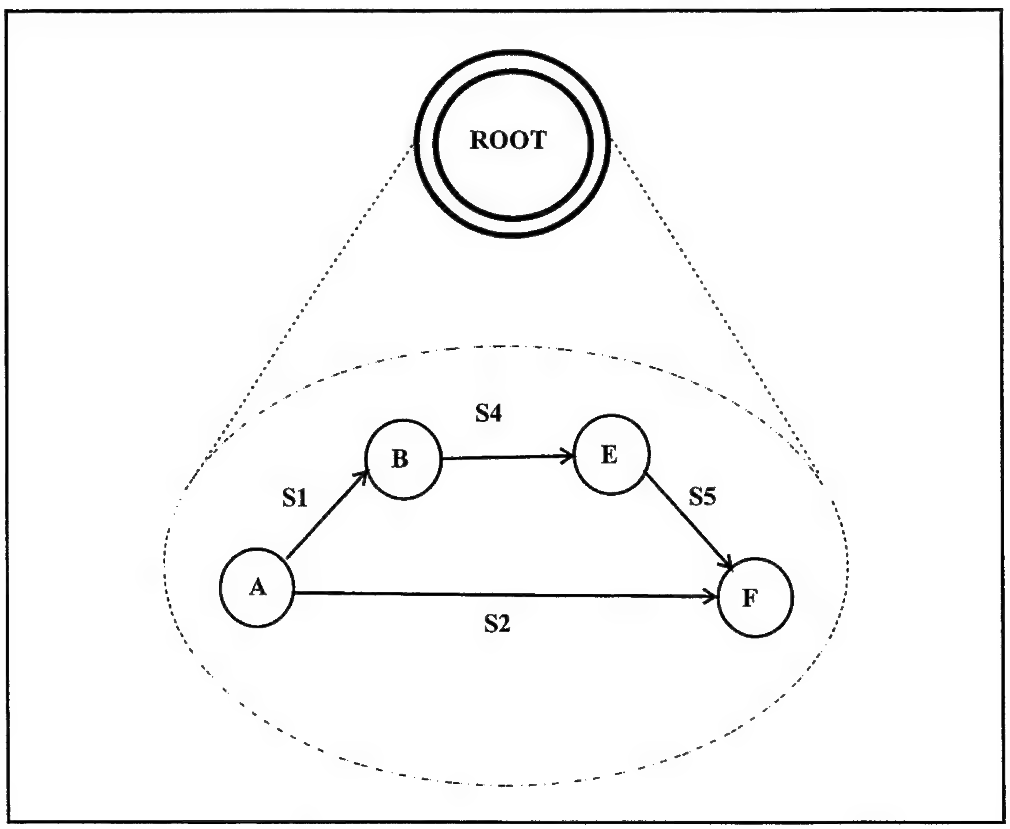

Figure 2.9: BASE Version of PSDL Prototype

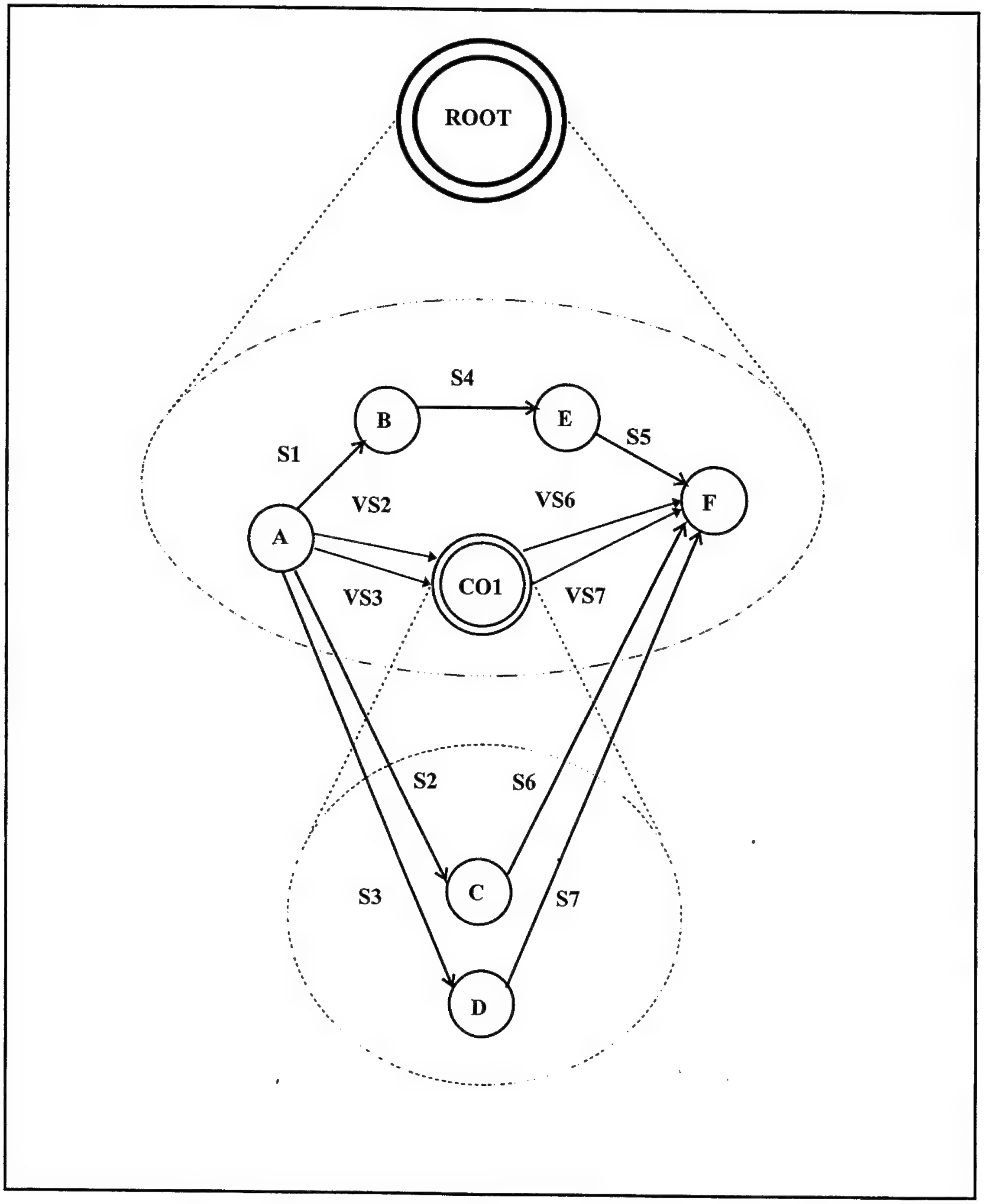

Figure 2.10: CHANGE A Version of PSDL Prototype

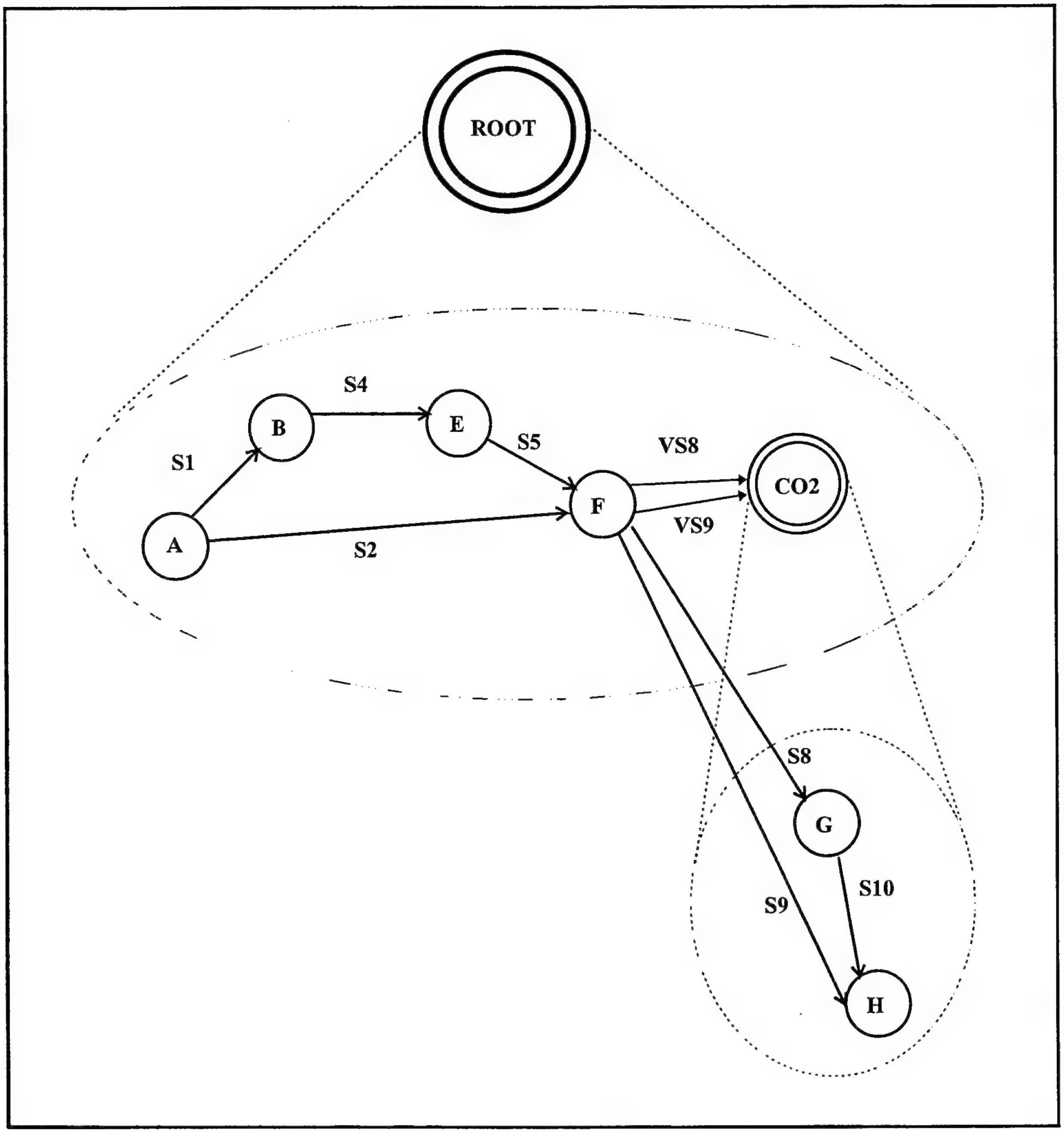

Figure 2.11: CHANGE B Version of PSDL Prototype

In sum, the **decompose_graph** algorithm can be viewed as a two stage process. The first stage recovers the parent-child relationship view of Figure 2.8 for the merged prototype. The second stage reconstructs the hierarchy of graphs, *decomposed* PSDL prototype of Figure 2.2 for the merged prototype based on the parent-child relationship view recovered in the first stage.

III. DECOMPOSITION RECOVERY STAGES

The current CAPS Change-Merge Tool takes three PSDL prototype versions as input, a BASE version, a CHANGE to the BASE version commonly called CHANGE A, and a second CHANGE to the BASE commonly called CHANGE B. The merge tool first produces a flattened version of each, which transforms each into a prototype that has one composite operator, ROOT, and one graph containing all atomic operators (see Figure 2.1). The merge tool next merges the specification parts of the three root operators. It then applies the slicing method to the corresponding graphs and merges the results. The result is a flattened merged prototype, commonly called MERGE, that has one merged composite operator, ROOT, and one merged graph containing all atomic operators. At this point, design decomposition structure recovery begins.

A. STAGE ONE: FINDING AND MERGING ANCESTOR CHAINS

As mentioned previously, automated decomposition recovery for a PSDL prototype can be viewed as a two stage process. The first stage involves automated recovery of the merged prototype's parent-child relationship view of Figure 2.8. The first step in this stage is to retrieve *ancestor chains* from BASE, CHANGE A, and CHANGE B for each atomic operator in MERGE. An ancestor chain is an ordered sequence of operator names which reflects the positional context of an operator in a decomposition structure [Ref. 1]. For example, C's ancestor chain in Figure 2.8 would be <ROOT, CO1>; F's would be <ROOT>. The function **find_ancestor_chain** in the **decompose_graph** algorithm finds and returns these ancestor chains [Ref. 1].

The next step in this stage is to merge the three recovered ancestor chains for each atomic operator in MERGE. The result is one merged ancestor chain per atomic operator. The theory that provides the capability to merge ancestor chains is developed and detailed

in [Ref. 1]. The end result of the theory is a formula for ancestor chain merge which has the formalized mathematical basis required for reliable automated software tools such as CAPS. This formula is detailed in [Ref. 1] and in Chapter IV of this thesis. The result of applying the merge formula to the CHANGE A, BASE, and CHANGE B ancestor chains for a given operator is a merged ancestor chain which at least approximates the positional context of the operator with regard to the changes to the prototype's decomosition structure, and in most practical cases exactly reflects the operator's positional context with regard to changes. The procedure in **decompose_graph** that performs ancestor chain merges is **merge_ancestor_chain**.

The theory developed in [Ref. 1] also provides the capability to automatically identify, report, and resolve ancestor chain merge conflicts. Conflicts generally result from differing positional contexts for a given operator in its recovered ancestor chains. An example would be an operator that has different parents in its CHANGE A and CHANGE B ancestor chains. Conflicts are resolved by taking the *greatest lower bound* of the conflicting chains and assigning this as the ancestor chain for the operator. The procedure in **decompose_graph** that identifies and reports conflicts is **report_conflicts**; the procedure that resolves conflicts is **resolve_conflicts**.

The end result of this first stage is a set of conflict-free merged ancestor chains, one for each operator in MERGE, which accurately reflects the significant decomposition structure of MERGE relative to CHANGE A, BASE, and CHANGE B versions of the prototype.

B. STAGE TWO: PROTOTYPE RECONSTRUCTION

The input to this stage is MERGE, pre-flattened CHANGE A, BASE, and CHANGE B, and the set of merged ancestor chains. The goal of this stage is to *decompose* MERGE.

The first step builds a skeletal *hierarchy of graphs* decomposition structure for MERGE based on the ancestor chains recovered in stage one. During this step, composite operators are created, their graphs are populated with vertices, corresponding edges, and associated timing and control constraints. The first step is complete when all required composite operators have been created and all atomic operators have been added to a composite operator's graph. Thus, the basic decomposition structure is in place, but composite operator specification and implementation parts are incomplete.

The last step in this stage finishes construction of composite operators. It involves determining input and output streams for composite operators, building *virtual* data streams, and filling out specifications. In some cases, values and attributes for new composite operators have to be retrieved from their namesakes in CHANGE A, BASE, and CHANGE B and merged to recover the value or attribute (the reason being that composite operators other than ROOT are destroyed in the merge, and thus are not available in MERGE). When this has been the case, the values and attributes are merged using the same algorithms used for these elements by the current Change-Merge Tool. The function in **decompose_graph** which reconstructs the prototype is **reconstruct_prototype**. See Chapter IV of this thesis for detail.

IV. DESIGN: DECOMPOSITION RECOVERY EXTENSION

This chapter details the design of the functions and procedures called in **decompose_graph** along with significant support functions, procedures, and abstract data types. The top level design for the Decomposition Recovery Extension to the CAPS Change-Merge Tool closely follows the **decompose_graph** algorithm as given in [Ref. 1] with the only significant difference relating to some of the data structures used. The design of **decompose_graph** has been allocated to three Ada packages: **decompose_graph_pkg**, **extended_ancestor_pkg**, and **reconstruct_prototype_utilities_pkg**.

In the remainder of this chapter, there is a Design Description section for each package with a sub-section for each significant module in the package. For each module, there is a brief functional overview, a Concrete Interface Specification, and an Algorithm Sketch. The Concrete Interface Specification and Algorithm Sketch are presented as figures which immediately follow each brief functional overview.

A. DESIGN: ADA PACKAGE DECOMPOSE_GRAPH_PKG

This package provides the external interface to the PSDL Decomposition Recovery Extension through the **decompose_graph** procedure call. The arguments to this procedure are 1) the **psdl_program** data structures corresponding to pre-expanded CHANGE A , BASE, and CHANGE B versions of the prototype, 2) MERGE, the flattened prototype that is the result of the merge of flattened (expanded) versions of CHANGE A , BASE, and CHANGE B, and 3) an empty **psdl_program** data structure that is used to return the reconstructed prototype.

Thus, given CHANGE A , BASE, CHANGE B, and MERGE versions of a PSDL prototype as input, this package returns a *decomposed* reconstructed prototype.

1. Module: decompose_graph

As mentioned above, **decompose_graph** provides the external interface to the PSDL Decomposition Recovery Sub System.

The **decompose_graph** algorithm presented in [Ref. 1] calls for merged chains to be stored in an array ANCESTOR of type **extended_ancestor**. In the following design, merged chains are stored in a map of *operator name* to ancestor chain where the ancestor chain is represented as a variable of type **extended_ancestor**. ANCESTORS is declared as type **ancestor_chains**; **ancestor_chains** is an instantiation of the *generic map* package. Also note that the arguments A_PSDL, BASE_PSDL, and B_PSDL are of type **psdl_program**, whereas the original algorithm calls for them to be of type **psdl_graph**. Also, for the following design, **decompose_graph** is a procedure instead of a function.

---

procedure decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE: in psdl_program;
NEW_PSDL: in out psdl_program);

Input:

A_PSDL: un-expanded version of prototype Change A
BASE_PSDL: un-expanded version of prototype BASE
B_PSDL: un-expanded version of prototype Change B
MERGE: expanded prototype that resulted from the merge of flattened versions of Change A, BASE, and Change B.
NEW_PSDL: empty psdl_program data structure used to return reconstructed prototype;

Output:

NEW_PSDL: reconstructed prototype complete with recovered decomposition structure.

Figure 4.1: Concrete Interface Specification for decompose_graph

```
Algorithm decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE: in psdl_program;
                          NEW_PSDL: in out psdl_program);

        ANCESTORS: ancestor_chains; -- map: operator name -> ancestor chain
        MERGE_CHAIN, A_CHAIN, BASE_CHAIN, B_CHAIN: extended_ancestor;
        root_op: psdl_id;
begin
        root_op: psdl_id := find_root(BASE);

        for each operator N in MERGE
        loop
                if N is an atomic operator then
                        A_CHAIN := find_ancestor_chain(N, root_op, A);
                        B_CHAIN := find_ancestor_chain(N, root_op, B);
                        BASE_CHAIN := find_ancestor_chain(N, root_op, BASE);
                        merge_ancestor_chains(A_CHAIN, BASE_CHAIN, B_CHAIN,
                                              MERGE_CHAIN);
                        bind N -> MERGE_CHAIN to ANCESTORS;
                endif;
        endloop;

        report_conflicts(ANCESTORS);
        resolve_conflicts(ANCESTORS);
        NEW_PSDL := reconstruct_prototype(MERGE, A_PSDL, BASE_PSDL, B_PSDL,
                                          ANCESTORS);

end decompose_graph;
```

Figure 4.2: Algorithm Sketch for decompose_graph

2. Module: find_ancestor_chain

This function is called three times for every atomic operator "N" in MERGE. The three calls recover N's ancestor chains from CHANGE A, CHANGE B, and BASE versions of the prototype.

Thus, for large, complex decomposition graphs, design and implementation of an efficient search algorithm for **find_ancestor_chain** is important. To facilitate the search for N's ancestor chain, use is made of a field in each operator's specification part named *parent* of type **psdl_component**. This field is a reference to the operator's immediate ancestor composite operator. The function **get_ancestor** returns the value of this field for a given operator.

function find_ancestor_chain(N, root_id: psdl_id; P: psdl_program) return extended_ancestor;

Input:

N: the operator's psdl_id name for which the chain will be recovered;
root_id: the root operator's psdl_id name as given in the merged prototype;
P: the PSDL prototype from which N's chain will be recovered;

Return Value:

N's ancestor chain recovered from P returned in type extended_ancestor;

Figure 4.3: Concrete Interface Specification for **find_ancestor_chain**

```
Algorithm find_ancestor_chain(N, root_op: psdl_id; P: psdl_program)
return extended_ancestor;

        ancestor: extended_ancestor;
        ancestor_id : psdl_id;

        Algorithm recover_chain(ancestor: extended_ancestor; operator_id,
                                root_ id: psdl_id; P: psdl_program)
        return psdl_id;
        begin
                If operator = root_ id then -- unwind the recursion
                        return root_op_id;
                else -- continue recursion
                        ancestor_id := recover_chain(ancestor,
                                get_ancestor(operator_id, P), root_id, P);
                        -- construct ancestor chain as recursion unwinds
                        append ancestor_id to ancestor -- the ancestor chain;
                        return operator_id;
                endif;
        end recover_chain;

begin -- find_ancestor_chain

        Initialize ancestor to empty;

        if N is not the root operator and N is an operator in P then
                -- recursively construct N's ancestor chain
                ancestor_id := recover_chain(ancestor,
                                        get_ancestor(N, P), root_op, P),
                append ancestor_id to ancestor -- N's recovered ancestor chain;
        endif;

        return ancestor;
end find_ancestor_chain;
```

Figure 4.4: Algorithm Sketch for **find_ancestor_chain**

3. Module: merge_ancestor_chains

merge_ancestor_chains is called to merge the ancestor chains recovered from un-expanded CHANGE A, BASE, and CHANGE B versions of the prototype.

The algorithm for the **merge_ancestor_chains** function applies the specific merge rules: (x[x]y = y = y[x]x), (y[x]y = y), and (y[y]y = y) first. This is an attempt to optimize the merge operation by handling the most common cases without having to resort to more involved merge processing required for the general case.

For the general case, *pseudo-difference*, *union*, and *intersection* operations are needed to perform the merge. Algorithms for these operations are given in the **extended_ancestor_pkg** section. Processing starts with taking the *pseudo-difference* of CHANGE A and the BASE, followed by the *intersection* operation applied to CHANGE A and CHANGE B, followed by taking the *pseudo-difference* of CHANGE B and the BASE. The terms that result from these operations are then combined in two separate *union* operations. Merge conflicts are indicated by a null **extended_ancestor** returned from the *union* operation. If conflict occurs, the conflicting chains are saved as an **improper_ancestor** data type. Conflict reporting and conflict resolution occur in subsequent processing.

With regard to merge conflicts, the algorithm for **merge_ancestor_chains** is based on the following observation: [A *pseudo-difference* Base] *union* [A *intersection* B] will never conflict given that [A *intersection* B] will always return a prefix of A and [A *pseudo-difference* Base] will either return A or **empty_ancestor**. The *union* of A with a prefix of A is A. The *union* of **empty_ancestor** with any prefix chain is the prefix chain. Thus, [A *pseudo-difference* Base] *union* [A *intersection* B] will never conflict. This implies that conflicts can only occur in the second *union* operation of the ancestor chain merge.

```
procedure merge_ancestor_chains(A_CHAIN, BASE_CHAIN, B_CHAIN: extended_ancestor;
                                MERGE_CHAIN: in out extended_ancestor);

Input:
        A_CHAIN: The ancestor chain recovered from Change A prototype version;
        BASE_CHAIN: The ancestor chain recovered from BASE prototype version;
        B_CHAIN: The ancestor chain recovered from Change B prototype version;
Output:
        MERGE_CHAIN: The result of applying the merge to A_CHAIN, BASE_CHAIN, and
                     B_CHAIN;
```

Figure 4.5: Concrete Interface Specification for **merge_ancestor_chains**

```
Algorithm merge_ancestor_chains(A_CHAIN, BASE_CHAIN, B_CHAIN:
                extended_ancestor; MERGE_CHAIN: in out extended_ancestor);

        a_pseudodiff_base, a_intersection_b, b_pseudodiff_base: extended_ancestor;
begin
        -- first try the simple cases
        if A_CHAIN = BASE_CHAIN then
                MERGE_CHAIN := build_proper_ancestor (B_CHAIN);
        else
                if B_CHAIN = BASE_CHAIN then
                        MERGE_CHAIN := build_proper_ancestor (A_CHAIN);
                else
                        if A_CHAIN = B_CHAIN then
                                MERGE_CHAIN := build_proper_ancestor (B_CHAIN);
                        else -- have to apply the merge formula
                                a_pseudodiff_base:= pseudo_difference(A_CHAIN, BASE_CHAIN);
                                a_intersection_b := intersection(A_CHAIN, B_CHAIN);
                                b_pseudodiff_base := pseudo_difference(B_CHAIN, BASE_CHAIN);
                                union_term := union(a_pseudodiff_base, b_pseudodiff_base);
                                MERGE_CHAIN := union(b_pseudodiff_base, union_term);
                                        if MERGE_CHAIN = null_ancestor then -- conflict
                                                MERGE_CHAIN :=
                                                        build_improper_ancestor(A_CHAIN,
                                                                                BASE_CHAIN,
                                                                                B_CHAIN);
                                        endif;
                        endif;
                endif;
        endif;
end merge_ancestor_chains;
```

Figure 4.6: Algorithm Sketch for **merge_ancestor_chains**

4. Module: report_conflicts

The algorithm for the **report_conflicts** procedure loops through the array of merged ancestor chains and outputs informative error messages for any conflicts that occurred during ancestor chain merge.

---

procedure report_conflicts(ea_map: ancestor_chains);

Input:

ea_map:a map of psdl_id operator name to proper and improper extended ancestors – the ancestor chains resulting from the merge operations;

Output:

a conflict message identifying conflicting terms of the merge operation;

Figure 4.7: Concrete Interface Specification for **report_conflicts**

```
Algorithm  report_conflicts(ea_map: ancestor_chains);
begin
        for every N psdl_id, extended_ancestor ea in ea_map loop
                if ea is an improper_ancestor then
                        put_conflict_message(N, ea);
                endif;
        end loop;
end report_conflicts;
```

Figure 4.8: Algorithm Sketch for **report_conflicts**

5. Module: resolve_conflicts

Conflicts arise when the *union* operation fails during the **merge_ancestor_chains** operation. This failure occurs when an operator's ancestor chains in the base and changed versions cannot be merged due to structural conflicts. For example, the result of the ancestor chain merge operation incorrectly requires an operator to have more than one immediate parent, which produces an improper ancestor lattice element [Ref. 1].

This improper lattice element is the *least upper bound* of two proper incomparable elements in the extended ancestor lattice domain and represents a merge conflict [Ref. 1]. The merge conflict is resolved by a call to **extended_ancestor_pkg.resolve_conflict** which replaces this improper element with the *greatest lower bound* of the conflicting merge terms.

---

procedure resolve_conflicts(ea_map: in out ancestor_chains);

Input:

ea_map: a map of psdl_id operator name to proper and improper extended ancestors – the ancestor chains resulting from the merge operations;

Output:

ea_map: a map psdl_id operator name to proper ancestor;

Figure 4.9: Concrete Interface Specification for **resolve_conflicts**

```
Algorithm  resolve_conflicts(ea_map: in out ancestor_chains);
        scan_ea_map: ancestor_chains;
begin
        scan_ea_map := Copy of ea_map;

        for every operator id and extended_ancestor ea in scan_ea_map loop
                if ea is an improper_ancestor then
                        ea := resolve_conflict(ea);
                        update the ea_map entry for id with proper_ancestor ea;
                endif;
        end loop;
end resolve_conflicts;
```

Figure 4.10: Algorithm Sketch for **resolve_conflicts**

6. Module: reconstruct_prototype

The arguments to this function are the original **psdl_program's** for the base and changed versions of the prototype, the **psdl_program** map for the merged prototype, and a map of merged ancestor chains (**extended_ancestor_records**), one map entry for each component in the merged prototype's **psdl_program**. The merged prototype's

psdl_program is composed of one root composite operator entry – all other entries are atomic operators. Given this as input, an algorithm for **reconstruct_prototype** follows.

The first section of the algorithm sets an access type to MERGE's root operator component and associated graph and then creates a root operator for NEW_PSDL with the name of MERGE's root operator.

The main section of the algorithm is a loop that constructs composite operators for each element in each atomic operator's recovered merged ancestor chain, builds a copy of each atomic operator and binds it to NEW_PSDL (the **psdl_program** for the reconstructed prototype), and then adds each atomic operator's attributes and properties to its parent composite operator. When this main section loop completes, the result is a partially reconstructed prototype decomposition structure in which all of the composite and atomic operators are in correct structural context, the atomic operators' specification and implementation parts are complete, but the composite operators have incomplete specification and implementation parts.

The last section of the algorithm calls a recursive procedure to finish up the composite operators' incomplete specification and implementation parts.

---

function reconstruct_prototype(MERGE, A, BASE, B: psdl_program,
ANCESTORS: ancestor_chains) return psdl_program;

Input:

A_PSDL: un-expanded version of prototype Change A
BASE_PSDL: un-expanded version of prototype BASE
B_PSDL: un-expanded version of prototype Change B
MERGE: expanded prototype that resulted from the merge of flattened versions of Change A, BASE, and Change B.
ANCESTORS: map of conflict-free merged ancestor chains

Returned Value:

Reconstructed prototype with recovered decomposition structure in a psdl_program data structure.

Figure 4.11: Concrete Interface Specification for **reconstruct_prototype**

```
Algorithm reconstruct_prototype(MERGE, A, BASE, B: psdl_program,
                                        ANCESTORS: ancestor_chains)
return psdl_program;

    NEW_PSDL: psdl_program;
    co_node, ancestor_node, new_root_op, merges_root_op: composite_operator;
    atomic_op: atomic_operator;
    root_id: psdl_id;
    chain: psdl_id_sequence;
    merges_graph, ancestor_graph, root_op_graph: psdl_graph;
    root_op_id, op, atomic_op_id: op_id;

begin

    NEW_PSDL := empty_psdl_program;

    root_id := MERGE's root operator psdl_id name;
    merges_root_op := MERGE's root operator psdl_component;
    merges_graph := MERGE's root operator psdl_component graph;
    new_root_op := make_composite_operator(root_id);

    bind root_id, new_root_op to NEW_PSDL;

    for every atomic_id: psdl_id, and ea: extended_ancestor in ANCESTORS
    loop
        get atomic_id's ancestor chain from ea;
        ancestor_node := new_root_op;

        for every chain_element in atomic_id's ancestor chain starting with root element
        loop
            if the composite operator for chain_element is already in NEW_PDSL then
                co_node := chain_element's component from NEW_PSDL;
            else
                co_node := make_composite_operator(chain_element);
                set co_node's parent to ancestor_node;

                -- add partial vertex definition for composite operator
                -- to the parent graph and try to retrieve vertex attributes
                -- from A, BASE, B entries for the composite.
                op := op_id identifier for co_node;
                add_composite_vertex(op, ancestor_node, A, BASE, B);

                bind chain_element, co_node to NEW_PSDL;
            endif;
            set ancestor_node to co_node for next iteration;
        endloop;
```

Figure 4.12: Algorithm Sketch for **reconstruct_prototype**

```
            atomic_op := make_atomic_operator(atomic_id's component from MERGE);

            set atomic_op's parent to ancestor_node;

            atomic_op_id := op_id identifier for atomic_op;

            ancestor_graph := copy of ancestor_node's graph;

            -- Call copy_vertex_n_edges to copy atomic_op_id's vertex & edges in merges_graph to
            -- ancestor_graph;
            copy_vertex_n_edges(atomic_op_id, merges_graph, ancestor_graph);

            -- Call copy_timer_operations to copy atomic_op_id's timer_operation entries in
            -- merges_root_op to ancestor_node;
            copy_timer_operations(atomic_op_id, merges_root_op, ancestor_node);

            -- Call copy_control_constraints to copy atomic_op_id's control constraint (output
            -- guards, exception triggers, execution guards, and triggers) entries
            -- in merges_root_op to ancestor_node;
            copy_control_constraints(atomic_op_id, merges_graph,
                                     merges_root_op, ancestor_node);


            -- Call copy_timing_constraints to copy atomic_op_id's timing constraints (periods,
            -- finished within's, minimum calling periods, and maximum response times)
            -- entries in merges_root_op to ancestor_node;
            copy_timing_constraints(atomic_op_id, merges_root_op, ancestor_node);

            bind atomic_id, atomic_op to NEW_PSDL;

      endloop;


-- At this point, a skeletal decomposition structure is in place - all of the composite operators are in place
-- with partially completed specification and implementation portions.
--
-- Next, finish up construction of the each composite operator in NEW_PSDL; input edges, output edges,
-- state edges, [smet's, exceptions,] initial states, and other attributes will have to be set in each composite
-- operator's specification and implementation part.
--
-- Starting with the root operator, call finish_composite_operator_construction to
-- recurse through composite operator graphs to finish reconstruction of each composite operator's
-- specification and implementation parts
      finish_composite_operator_construction(graph(new_root_op), A, BASE, B,
                                             NEW_PSDL, new_root_op, new_root_op);

      return NEW_PSDL;
end reconstruct_prototype;
```

Figure 4.13: Algorithm Sketch for **reconstruct_prototype** (cont.)

B. DESIGN: ADA PACKAGE EXTENDED_ANCESTOR_PKG

Package **extended_ancestor_pkg** provides type **extended_ancestor**, the basic manipulation functions for this type, and the *union, intersection, pseudo-difference* operations used in the ancestor chain merge operation (**merge_ancsetor_chains**).

In the descriptions which follow, the designs of most of the functions and procedures that make up the public interface to this package are described in detail. The designs of some of the more interesting local support functions and procedures are described in detail as well. But, for most of the trivial support functions and procedures, a brief statement of purpose is given followed by a Concrete Interface Specification.

1. Type Extended Ancestor

Type **extended ancestor** is designed as an Ada private type. It is essentially a data structure used to store *proper* and *improper ancestor chain sequences*. Ancestor chain sequences are ordered sequences of PSDL composite operator names. The first element in the chain is the name of a prototype's root operator. Each subsequent element in the chain is the child of its immediate predecessor, and the last element is the name of an atomic operator' immediate parent composite operator.

A *proper* ancestor is an element of the set of all finite sequences partially ordered by the prefix ordering [Ref. 1]. In the following design, **proper_ancestor** is an access type for an **extended_ancestor_record** with discriminant **ancestor => proper**. This subtype is used to store an atomic operator's properly formed ancestor chain as a sequence of **psdl_id** names of composite operators.

An *improper* ancestor is an improper data element representing a *least upper bound* for a set of incomparable proper elements in the extended ancestor lattice [Ref. 1]. In the following design, **improper_ancestor** is a pointer to an

extended_ancestor_record with discriminant **ancestor => improper**. This subtype is used to store conflicting proper ancestor chains for subsequent conflict reporting and resolution.

```
-- Discriminant for type extended_ancestor_record
type ancestor_type is (proper, improper);

-- storage for both "proper" and "improper" ancestor chains.
type extended_ancestor_record
        (ancestor: ancestor_type)
is private;

type extended_ancestor is access extended_ancestor_record;

subtype proper_ancestor is extended_ancestor(ancestor => proper);

subtype improper_ancestor is extended_ancestor(ancestor =>improper);

null_ancestor: constant extended_ancestor := null;

empty_extended_ancestor: extended_ancestor;

-- raised when null_ancestor is unexpectedly encountered.
undefined_ancestor: exception;

-- raised when an undefined ancestor chain is unexpectedly encountered.
undefined_ancestor_chain: exception;

-- raised when comparison of an improper-to-proper ancestor is unexpectedly
-- attempted
ancestor_type_mismatch: execption;
```

Figure 4.14 :Concrete Interface Specification for **Type Extended Ancestor**

2. Module: greatest_common_prefix

Function **greatest_common_prefix** finds and returns the greatest common prefix (*greatest lower bound*) of two proper ancestor chain sequences. It is a local support function, but it is described in detail here to afford the reader a better understanding of the **intersection**, **put_conflict_message**, and **resolve_conflict** functions.

function greatest_common_prefix(chain_1, chain_2: psdl_id_sequence) return psdl_id_sequence;

Input:

chain_1: psdl_id_sequence representing an ancestor chain
chain_2: psdl_id_sequence representing an ancestor chain

Return Value:

The greatest common prefix or chain_1 and chain_2 returned in a psdl_id_sequence data structure

Figure 4.15: Concrete Interface Specification for **greatest_common_prefix**

```
Algorithm greatest_common_prefix(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence;
        compare_limit: natural;
        result: psdl_id_sequence;
        I: natural := 1;
        elements_match: Boolean := True;
begin
        Initialize the return sequence "result" to empty;

        -- Find and set the range for chain element comparison;
        if the length of chain_1 is greater than the length of chain_2 then
                compare_limit := length of chain_2;
        else
                compare_limit := length of chain_1;
        endif;
        -- extract the greatest common prefix and store it in "result"
        while I is less than or equal to compare_limit and elements_match loop
                if chain_1 element  I  equals chain_2 element  I then
                        add the element to the result sequence;
                        I := I + 1;
                else
                        elements_match := False;
                end if;
        end loop;
        return result;
end greatest_common_prefix;
```

Figure 4.16: Algorithm Sketch for **greatest_common_prefix**

3. Module: is_prefix_of

This is a local overloaded function used to determine if the first **extended_ancestor** argument is a prefix of the second (or if the first ancestor chain sequence argument is a prefix of the second). An ancestor chain sequence AC1 of length L1 is a prefix of ancestor chain sequence AC2 of length L2 if each element of AC1, beginning with the first element up to and including element L1, matches each corresponding element of AC2, beginning with the first element up to and including element L1 of AC2.

Although **is_prefix_of** is a local support function, it is described in detail here to afford the reader a better understanding of the **intersection**, **union**, **pseudo_difference**, **put_conflict_message**, and **resolve_conflict** functions.

---

function is_prefix_of(chain_1, chain_2: psdl_id_sequence) return boolean;

Input:

chain_1: psdl_id_sequence of operator names representing an ancestor chain
chain_2: psdl_id_sequence of operator names representing an ancestor chain

Return value:

Boolean True if chain_1 is a prefix of chain_2, Boolean False otherwise

function is_prefix_of(ea_1, ea_2: extended_ancestor) return boolean;

Input:

ea_1: extended_ancestor access type for a proper ancestor
ea_2: extended_ancestor access type for a proper ancestor

Return value:

Boolean True if ea_1 is a prefix of ea_2, Boolean False otherwise

---

Figure 4.17: Concrete Interface Specification for **is_prefix_of** (overloaded)

```
function is_prefix_of(chain_1, chain_2: psdl_id_sequence)
return boolean;
        is_prefix: Boolean := False;
begin
        if length of chain_1 is greater than length of chain_2 then -- can't be prefix of shorter chain
                is_prefix := False;
        else
                gets the prefix slice of chain_2 from the first element to the length of chain_1;

                if chain_1 equals prefix slice of chain_2 then
                        is_prefix := True;
                else
                        is_prefix := False;
                end if;
        end if;

        return is_prefix;

end is_prefix_of;

function is_prefix_of(ea_1, ea_2: extended_ancestor)
return boolean
        return (is_prefix_of(ea_1.chain, ea_2.chain));
end is_prefix_of;
```

Figure 4.18: Algorithm Sketch for **is_prefix_of** (overloaded)

4. Module: intersection

This overloaded function has both a public and local version. The public version function performs the *intersection* operation on two arguments of type **extended_ancestor**. The local version performs the *intersection* operation on to arguments of type **psdl_id_sequence**. Both versions return the greatest common prefix for two supplied arguments.

The domain of the operation is the extended ancestor lattice. In this extended domain, the *intersection* operation is essentially the *set intersection* operation [Ref. 1]. The general rule for the *intersection* operation in the extended ancestor lattice domain is:

```
EA1, EA2: extended_ancestor;
if EA1 is a prefix of EA2 then
        intersection(EA1, EA2) = EA1;
else
        if EA2 is a prefix of EA1 then
                intersection(EA1, EA2) = EA2;
        else
                intersection(EA1, EA2) = greatest_common_prefix(EA1, EA2);
        endif;
endif;
```

This rule applies to **psdl_id_sequences** ancestor chain sequences as well.

The following algorithms for *intersection* first check to see whether one of the **extended_ancestor** (or **psdl_id_sequence** ancestor chain) arguments is a prefix of the other, and if so, return a copy of the prefix argument. Otherwise, the algorithms find and return the greatest common prefix of the two arguments.

---

function intersection(ea_1, ea_2: extended_ancestor) return extended_ancestor;

Input:

ea_1: extended_ancestor access type for a proper ancestor
ea_2: extended_ancestor access type for a proper ancestor

Return Value:

extended_ancestor access type for result of the intersection operation applied to ea_1 and ea_2

function intersection(chain_1, chain_2: psdl_id_sequence) return psdl_id_sequence;

Input:

chain_1: psdl_id_sequence of operator names representing an ancestor chain
chain_2: psdl_id_sequence of operator names representing an ancestor chain

Return Value:

The result of the intersection operation applied to chain_1 and chain_2 in a psdl_id_sequence of operator names representing an ancestor chain

---

Figure 4.19: Concrete Interface Specification for **intersection** (overloaded)

```
Algorithm intersection(ea_1, ea_2: extended_ancestor)
return extended_ancestor;
        result: extended_ancestor;
begin
        if  is_prefix_of(ea_1, ea_2) then
                result := build_proper_ancestor( ea_1.chain);
        else
                if is_prefix_of (ea_2, ea_1) then

                        result := build_proper_ancestor( ea_2.chain);
                else
                        result := build_proper_ancestor(
                                        greatest_common_prefix (ea_2.chain, ea_1.chain));
                endif;
        endif;
        return result;
end intersection;

Algorithm intersection(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence;
        result: psdl_id_sequence;
begin
        if  is_prefix_of(chain_1, chain_2) then
                result := chain_1;
        else
                if is_prefix_of (chain_2, chain_1) then

                        result := chain_2;
                else
                        result :=  greatest_common_prefix (chain_2, chain_1);
                endif;
        endif;
        return result;
end intersection;
```

Figure 4.20: Algorithm Sketch for **intersection** (overloaded)

5. Module: pseudo_difference

This overloaded function has both a public and local version. The public version performs the Brouwerian Algebra *pseudo-difference* operation on two arguments of type **extended_ancestor**. The local version performs the Brouwerian Algebra *pseudo-difference* operation on to arguments of type **psdl_id_sequence**. Both versions return the *pseudo-difference* of the two supplied arguments.

The Brouwerian Algebra *pseudo-difference* applied to a pair of **extended_ancestor** (or **psdl_id_sequence**) arguments is essentially a *set difference* followed by a *downward closure* operation applied to the result. [Ref. 1]

The general rule for this operation in the extended ancestor lattice domain is:

if EA1 is a prefix of EA2 then
        pseudo_difference(EA1, EA2) = empty_extended_ancestor;
else
        pseudo_difference(EA1, EA2) = EA1;
endif;

This rule applies to **psdl_id_sequences** ancestor chain sequences as well.

The following algorithm for function **pseudo_difference** first checks to see whether the first argument is a prefix of the other, and if so, returns an empty **extended_ancestor**. Otherwise, the algorithm returns the first argument.

---

function pseudo_difference(ea_1, ea_2: extended_ancestor) return extended_ancestor;

Input:
ea_1: extended_ancestor access type for a proper ancestor
ea_2: extended_ancestor access type for a proper ancestor

Return Value:
extended_ancestor access type for result of the pseudo-difference operation applied to ea_1 and ea_2

function pseudo_difference(chain_1, chain_2: psdl_id_sequence) return psdl_id_sequence;

Input:
chain_1: psdl_id_sequence of operator names representing an ancestor chain
chain_2: psdl_id_sequence of operator names representing an ancestor chain

Return Value:
The result of the pseudo-difference operation applied to chain_1 and chain_2 in a psdl_id_sequence of operator names representing an ancestor chain

---

Figure 4.21: Concrete Interface Specification for **pseudo_difference**

```
Algorithm pseudo_difference(ea_1, ea_2: extended_ancestor)
return extended_ancestor;
begin
        if is_prefix_of(ea_1, ea_2) then
                return an empty extended_ancestor;
        else
                return a copy of ea_1;
        endif;
end pseudo_difference;


Algorithm pseudo_difference(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence;
begin
        if is_prefix_of(chain_1, chain_2) then
                return an empty psdl_id_sequence;
        else
                return a copy of chain_1;
        endif;
end pseudo_difference;
```

Figure 4.22: Algorithm Sketch for **pseudo_difference**

6. Module: union

This overloaded module has both a public and local version. The public version function performs the *union* operation on two arguments of type **extended_ancestor**. The local version procedure performs the *union* operation on to arguments of type **psdl_id_sequence**. Both versions return the result of the *union* operation as applied to the two supplied arguments.

The *union* operation forms the *least upper bound* for two **extended_ancestor** (or two **psdl_id_sequence** ancestor chain) arguments in the extended ancestor lattice domain. The general rule for this operation in this domain is:

EA1, EA2: extended_ancestor;
if EA1 is a prefix of EA2 then
    union(EA1, EA2) = EA2;
else
    CONFLICT -- UNDEFINED;
endif;

This rule applies to **psdl_id_sequences** ancestor chain sequences as well.

The following algorithm implements the above general rule. It first checks for empty **extended_ancestor** arguments and attempts to return a non-empty **extended_ancestor**. If both of the **extended_ancestor** arguments are empty, then an empty **extended_ancestor** is returned. If the **extended_ancestor** arguments are non-empty, the algorithm checks to see whether one argument is a prefix of the other, and if so, returns the other argument. In the cases mentioned so far, the *union* operation is successful, and the returned **extended_ancestor** is either empty or non-empty.

The remaining case is the conflict case – the algorithm is unable to form an *union* and will return a null **extended_ancestor** indicating an undefined result. In the context of ancestor chain merge operations, an *undefined* result for an *union* operation signals a merge conflict.

function union(ea_1, ea_2: extended_ancestor) return extended_ancestor;

Input:

ea_1: extended_ancestor access type for a proper ancestor
ea_2: extended_ancestor access type for a proper ancestor

Return Value:

extended_ancestor access type for result of the union operation applied to ea_1 and ea_2

procedure union(chain_1, chain_2: psdl_id_sequence;
result: in out psdl_id_sequence;
conflict: in out Boolean);

Input:

chain_1: psdl_id_sequence of operator names representing an ancestor chain
chain_2: psdl_id_sequence of operator names representing an ancestor chain
result: empty psdl_id_sequence used to return result of union operation
conflict: Boolean variable used to signal conflict

Output:

result: psdl_id_sequence containing result of union operation as applied to chain_1 and chain_2
conflict: Boolean variable set to True of conflict occurred, False otherwise

Figure 4.23: Concrete Interface Specification for **union** (overloaded)

```
Algorithm union(ea_1, ea_2: extended_ancestor) return extended_ancestor;
        result: extended_ancestor;
begin

        if ea_1 is an empty extended_ancestor then
                result :=  copy of ea_2;
        else    if ea_2 is an empty extended_ancestor then
                        result := copy of ea_1;
                else    if is_prefix_of (ea_1, ea_2) then
                                result := copy of ea_2;
                        else    if is_prefix_of(ea_2,  ea_1) then
                                        result := copy of ea_1;
                                else -- can't form a union
                                        result := null_ancestor;
                                endif;
                        endif;
                endif;
        endif;

        return result;

end union;

Algorithm union(chain_1, chain_2: psdl_id_sequence;
                result: in out psdl_id_sequence;
                conflict: in out Boolean);
begin
        conflict := False;
        if is_prefix_of (chain_1, chain_2) then
                result := copy of chain_2;
        else
                if is_prefix_of(chain_2, chain_1) then
                        result := copy of chain_1;
                else -- can't form a union
                        conflict := True;
                endif;
        endif;
end union;
```

Figure 4.24: Algorithm Sketch for **union** (overloaded)

7. Module: resolve_conflict

This function takes an **improper_ancestor** resulting from a merge conflict as input, reconstructs the merge conflict, resolves the conflict, and returns the conflict-free result in a newly constructed **proper_ancestor**.

The algorithm for **resolve_conflict** is based on the following observation: [A *pseudo-difference* Base] *union* [A *intersection* B] will never conflict given that [A *intersection* B] will always return a prefix of A and [A *pseudo-difference* Base] will either return A or empty_ancestor. The *union* of A with a prefix of A is A. The *union* of **empty_ancestor** with any prefix chain is the prefix chain. Thus, [A *pseudo-difference* Base] *union* [A *intersection* B] will never conflict.

This implies that conflicts will only occur in the second *union* operation of the ancestor chain merge.

---

function resolve_conflict(ia: improper_ancestor) return proper_ancestor;

Input:

ia: improper_ancestor resulting from a merge conflict;

Return Value:

proper_ancestor with ancestor chain that resulted from conflict resolution

---

Figure 4.25: Concrete Interface Specification for **resolve_conflict**

```
Algorithm resolve_conflict(ia: improper_ancestor) return proper_ancestor;
        gcp, union_term, a_pseudodiff_base,
        a_intersection_b, b_pseudodiff_base: psdl_id_sequence;

        resolved_chain: proper_ancestor;
begin

        -- reconstruct the 3 terms from the conflicting merge
        a_pseudodiff_base := pseudo_difference(ia.chain_A, ia.chain_BASE);
        a_intersection_b := intersection(ia.chain_A, ia.chain_B);
        b_pseudodiff_base := pseudo_difference(ia.chain_B, ia.chain_BASE);

        -- rebuild a_pseudodiff_base U a_intersection_b term
        union(a_pseudodiff_base, a_intersection_b, union_term, conflict);

        -- find the proper common prefix of the 2 conflicting terms
        -- union_term U b_pseudodiff_base
        gcp := greatest_common_prefix(union_term, b_pseudodiff_base);

        resolved_chain := build_proper_ancestor(gcp);

        return resolved_chain;

end resolve_conflict;
```

Figure 4.26: Algorithm Sketch for **resolve_conflict**

8. Module: put_conflict_message

Procedure put_conflict_message takes an **improper_ancestor** as input, reconstructs the merge conflict, and outputs an informative message detailing the conflict in reasonable depth.

The same observation given in the Algorithm Sketch for Module **resolve_conflict** applies to the algorithm for **put_conflict_message**. Refer to the Algorithm Sketch for Module **resolve_conflict** for detail.

---

procedure put_conflict_message(N: psdl_id; ia: improper_ancestor);

Input:

N: the psdl_id name for the atomic operator whose improper ancestor chain is represented by the next argument

a: N's improper ancestor chain

Output:

informative message detailing merge conflict

Figure 4.27: Concrete Interface Specification for **put_conflict_message**

```
Algorithm put_conflict_message(N: psdl_id; ia: improper_ancestor);
        gcp, union_term, union_term_imp,a_pseudodiff_base,
        a_intersection_b, b_pseudodiff_base, b_pseudodiff_base_imp: psdl_id_sequence;

        gcp_len: natural := 0;

        conflict: Boolean := False;
begin

        -- reconstruct the 3 terms for the conflicting merge
        a_pseudodiff_base := pseudo_difference(ia.chain_A, ia.chain_BASE);
        a_intersection_b := intersection(ia.chain_A, ia.chain_B);
        b_pseudodiff_base := pseudo_difference(ia.chain_B, ia.chain_BASE);

        -- rebuild a_pseudodiff_base U a_intersection_b term
        union(a_pseudodiff_base, a_intersection_b, union_term, conflict);

        -- find the proper common prefix of the 2 conflicting terms
        -- union_term U b_pseudodiff_base
        gcp := greatest_common_prefix(union_term, b_pseudodiff_base);

        -- find the improper elements of the 2 conflicting terms
        union_term_conflict_slice := union_term -gcp
        b_pseudodiff_base_conflict_slice := b_pseudodiff_base - gcp;

        put("ONE OR MORE CONFLICTS IN ANCESTOR CHAIN RECOVERY FOR: ");
        put_line(convert(N));
        put("<"); put_chain(ia.chain_A, False); put_line(">");
        put("[<"); put_chain(ia.chain_BASE, False);  put_line(">]");
        put("<"); put_chain(ia.chain_b, False); put_line("> =");

        put("<"); put_chain(union_term, False);
        put_line("> U ");
        put("<"); put_chain(b_pseudodiff_base, False);
        put_line(">=");
        put_line("(***conflict***)=");
        put("<"); put_chain(gcp, False);
        put("(");put_chain(union_term_imp, False);
        put(" U ");
        put_chain(b_pseudodiff_base_imp, False); put_line(")>");
        put_line("");

end put_conflict_message;
```

Figure 4.28: Algorithm Sketch for **put_conflict_message**

9. Support Functions and Procedures for extended_ancestor_pkg

The modules described below are sufficiently trivial as not to warrant detailed description. Refer to the source listings for Package **extended_ancestor_pkg** in Appendix A for detail.

Module type_of_ancestor

Purpose:
returns an extended_ancestor's discriminant: "proper" or "improper"

Concrete Interface Specification:
function type_of_ancestor(ea: extended_ancestor) return ancestor_type;

Module empty_ancestor

Purpose:
returns a proper ancestor with an empty ancestor chain

Concrete Interface Specification:
function empty_ancestor return proper_ancestor;

Module append_ancestor

Purpose:
appends a operator's psdl_id name to an extended_ancestor's ancestor chain

Concrete Interface Specification:
procedure append_ancestor(ea: in out extended_ancestor; ancestor_id: psdl_id);

Module assign_chain

Purpose:
assigns proper_ancestor *ea_2*'s ancestor chain to proper_ancestor *ea_1*; recycles *ea_2*'s existing ancestor chain prior to new assignment.

Concrete Interface Specification:
procedure assign_chain(ea_1: in out proper_ancestor; ea_2: proper_ancestor);

Module assign_chain

Purpose:
assigns psdl_id_sequence *chain* as proper_ancestor *ea*'s ancestor chain; recycles *ea*'s existing ancestor chain prior to new assignment.

Concrete Interface Specification:
procedure assign_chain(ea: in out proper_ancestor; chain: psdl_id_sequence);

Figure 4.29: Support Functions and Procedures for extended_ancestor_pkg

Module build_proper_ancestor

Purpose:
returns a proper ancestor initialized to the supplied ancestor chain sequence

Concrete Interface Specification:
function build_proper_ancestor(ea_chain: psdl_id_sequence) return proper_ancestor;

Module build_improper_ancestor

Purpose:
returns an improper ancestor initialized to the supplied ancestor chain sequences

Concrete Interface Specification:
function build_improper_ancestor(a_chain, base_chain, b_chain:
psdl_id_sequence) return improper_ancestor;

Module eq

Purpose:
determines equality for both proper_ancestor's and improper_ancestor's

Concrete Interface Specification:
function eq(ea_1, ea_2: extended_ancestor) return boolean;

Module recycle_extended_ancestor

Purpose:
recycle storage for proper or improper extended_ancestor_records

Concrete Interface Specification:
procedure recycle_extended_ancestor(ea: in out extended_ancestor);

Module get_ancestor

Purpose:
get the psdl_id identifier of a component's ancestor

Concrete Interface Specification:
function get_ancestor(id: psdl_id; p: psdl_program) return psdl_id;

Module get_chain

Purpose:
return a proper ancestor's psdl_id_sequence ancestor chain

Concrete Interface Specification:
function get_chain(ea: extended_ancestor) return psdl_id_sequence;

Figure 4.30: Support Functions and Procedures for **extended_ancestor_pkg** (cont.)

C. DESIGN: ADA PACKAGE RECONSTRUCT_PROTOTYPE_UTILITIES_PKG

The **reconstruct_prototype_utilities_pkg** package provides the utility functions and procedures used by **decompose_graph_pkg** function **reconstruct_prototype** to reconstruct a PSDL prototype's decomposition structure.

A number of modules in this package were taken from [Ref. 2]. For these modules, a brief statement of purpose and Concrete Interface Specification is given. In cases where these modules have been altered, the alteration is noted and briefly explained.

1. Module: merge_output_stream_type_names

This procedure merges output stream type names recovered from original **CHANGE A**, **BASE**, and **CHANGE B** prototype composite operators for use in the post-merge reconstruction of new composite operators during decomposition recovery. It is necessary to go the original **CHANGE A**, **BASE**, and **CHANGE B** prototypes to get the output stream type names given that they are lost in the pre-merge flattening process, and thus are absent from the flattened merged prototype.

---

procedure merge_output_stream_type_names(merged_type_name: in out type_name;
id, stream_name: psdl_id;
A, BASE, B: psdl_program);

Input:

merged_type_name: used to return the merged type_name
id: psdl_id name of composite operator for which merge will be accomplished
stream_name: the name of the output stream for which the type_name will be merged
A: pre-merge version of Change A prototype
BASE: pre-merge version of BASE prototype
B: pre-merge version of Change B prototype

Output:

merged_type_name: merged type_name for output stream

Figure 4.31: Concrete Interface Specification for **merge_output_stream_type_names**

```
Algorithm merge_output_stream_type_names(merged_type_name: in out type_name;
                                          id, stream_name: psdl_id;
                                          A, BASE, B: psdl_program);

        a_name, base_name, b_name: type_name := null_type;
        op: composite_operator;
begin
        if operator "id" is a member of prototype A then
                fetch operator "id" from prototype A;
                if "stream_name" is an output stream for operator "id" then
                        a_name := type_name of "stream_name";
                endif;
        endif;

        if operator "id" is a member of prototype BASE then
                fetch operator "id" from prototype BASE;
                if "stream_name" is an output stream for operator "id" then
                        base_name := type_name of "stream_name";
                endif;
        endif;

        if operator "id" is a member of prototype B then
                fetch operator "id" from prototype B;
                if "stream_name" is an output stream for operator "id" then
                        b_name := type_name of "stream_name";
                endif;
        endif;

        merged_type_name := merge_types(base_name, a_name, b_name);

end merge_output_stream_type_names;
```

Figure 4.32: Algorithm Sketch for **merge_output_stream_type_names**

2. Module: merge_input_stream_type_names

This procedure merges input stream type names recovered from original **CHANGE A**, **BASE**, and **CHANGE B** prototype composite operators for use in the post-merge reconstruction of new composite operators during decomposition recovery. It is necessary to go the original **CHANGE A**, **BASE**, and **CHANGE B** prototypes to get the input stream type names given that they are lost in the pre-merge flattening process, and thus are absent from the flattened merged prototype.

procedure merge_input_stream_type_names(merged_type_name: in out type_name;
id, stream_name: psdl_id;
A, BASE, B: psdl_program);

Input:

merged_type_name: used to return the merged type_name
id: psdl_id name of composite operator for which merge will be accomplished
stream_name: the name of the input stream for which the type_name will be merged
A: pre-merge version of Change A prototype
BASE: pre-merge version of BASE prototype
B: pre-merge version of Change B prototype

Output:

merged_type_name: merged type_name for input stream

Figure 4.33: Concrete Interface Specification for **merge_input_stream_type_names**

```
Algorithm merge_input_stream_type_names(merged_type_name: in out type_name;
                                        id, stream_name: psdl_id;
                                        A, BASE, B: psdl_program);
        a_name, base_name, b_name: type_name := null_type;
        op: composite_operator;
begin

        if operator "id" is a member of prototype A then
                fetch operator "id" from prototype A;
                if "stream_name" is an input stream for operator "id" then
                        a_name := type_name of "stream_name";
                endif;
        endif;

        if operator "id" is a member of prototype BASE then
                fetch operator "id" from prototype BASE;
                if "stream_name" is an input stream for operator "id" then
                        base_name := type_name of "stream_name";
                endif;
        endif;

        if operator "id" is a member of prototype B then
                fetch operator "id" from prototype B;
                if "stream_name" is an input stream for operator "id" then
                        b_name := type_name of "stream_name";
                endif;
        endif;

        merged_type_name := merge_types(base_name, a_name, b_name);

end merge_input_stream_type_names;
```

Figure 4.34: Algorithm Sketch for **merge_input_stream_type_names**

3. Module: merge_vertex_attributes

This procedure merges *maximum execution times* and *vertex properties* recovered from original **CHANGE A, BASE,** and **CHANGE B** prototype composite operators for use in the post-merge reconstruction of new composite operators during decomposition recovery. It is necessary to go the original **CHANGE A, BASE**, and **CHANGE B** prototypes to get the vertex attributes given that they are lost in the pre-merge flattening process and thus are absent from the flattened merged prototype.

---

procedure merge_vertex_attributes(merged_met: in out millisec;
vertex_properties: in out init_map;
op: op_id; co_name: psdl_id;
A, BASE, B: psdl_program);

Input:

merged_met: used to return merges met
vertex_properties: used to return merged vertex properties
op: op_id identifier for composite operator for which merge will be accomplished
co_name: psdl_id identifier for composite operator for which merge will be accomplished
A: pre-merge version of Change A prototype
BASE: pre-merge version of BASE prototype
B: pre-merge version of Change B prototype

Output:

merged_met: merged met for composite operator
vertex_properties: merged vertex properties for composite operator

---

Figure 4.35: Concrete Interface Specification for **merge_vertex_attributes**

```
Algorithm merge_vertex_attributes(merged_met: in out millisec;
                                  vertex_properties: in out init_map;
                                  op: op_id; co_name: psdl_id;
                                  A, BASE, B: psdl_program);

        a_graph, base_graph, b_graph: psdl_graph;
        a_diff_base, a_int_b, b_diff_base, a_met, base_met, b_met: millisec := undefined_time;
begin
        initialize a_graph, base_graph, and b_graph to empty;

        if co_name is an operator in prototype A then
                a_graph := copy of co_name's graph from A
                if op is a vertex in a_graph then
                        a_met := met value of op in a_graph;
                end if;
        endif;

        if co_name is an operator in prototype BASE then
                base_graph := copy of co_name's graph from BASE
                if op is a vertex in base_graph then
                        base_met := met value of op in base_graph;
                end if;
        endif;

        if co_name is an operator in prototype B then
                b_graph := copy of co_name's graph from B
                if op is a vertex in b_graph then
                        b_met := met value of op in b_graph;
                end if;
        endif;

        -- Taken from [3]. Note that in [3], system.max_int is assigned instead of undefined_time;
        if a_met <= b_met then a_int_b := b_met; else a_int_b := a_met; endif;

        if base_met <= a_met then a_diff_base := undefined_time; else a_diff_base := a_met; endif;

        if base_met <= b_met then b_diff_base := undefined_time; else b_diff_base := b_met; endif;
```

Figure 4.36: Algorithm Sketch for **merge_vertex_attributes**

```
if a_diff_base <= a_int_b then
        if a_diff_base <= b_diff_base then
                merged_met := a_diff_base;
        else
                merged_met := b_diff_base;
        endif;

else
        if a_int_b <= b_diff_base then
                merged_met := a_int_b;
        else
                merged_met := b_diff_base;
        endif;
endif;

-- Now, based on which prototype the met was recovered from, get
-- the corresponding vertex_property init_map.

if merged_met = base_met and op is a vertex in base_graph then
        vertex_properties := copy of op's vertex_properties from base_graph;
elsif merged_met = a_met and op is a vertex in a_graph then
        vertex_properties := copy of op's vertex_properties from a_graph;
elsif merged_met = b_met and op is a vertex in b_graph then
        vertex_properties := copy of op's vertex_properties from b_graph;
else
        vertex_properties := empty_init_map;
end if;

recycle a_graph, base_graph, and b_graph;

end merge_vertex_attributes;
```

Figure 4.37: Algorithm Sketch for **merge_vertex_attributes** (cont.)

4. Module: add_composite_vertex

This module is used to create a composite vertex and add it to a composite operator's graph. The vertex attributes are merged from the corresponding attributes in the un-expanded prototypes **CHANGE A**, **BASE**, and **CHANGE B**.

procedure add_composite_vertex(v: op_id; co: in out composite_operator; A, BASE, B: psdl_program);

Input:

V: op_id identifier for vertex to add to composite operator
co: composite operator the vertex v will be added to
A: pre-merge version of Change A prototype
BASE: pre-merge version of BASE prototype
B: pre-merge version of Change B prototype

Output:

co: composite operator co with update graph

Figure 4.38: Concrete Interface Specification for **add_composite_vertex**

Algorithm add_composite_vertex(v: op_id; co: in out composite_operator; A, BASE, B: psdl_program);

```
        co_graph: psdl_graph;
        op: psdl_component;
        vertex_properties: init_map;
        merged_met: millisec := undefined_time;
begin

        -- call merge_vertex_attributes  to merge v's met and vertex properties
        -- from the definitions of co in A, BASE, B prototypes given that these
        -- values are unavailable in the flattened merged prototype; return thes
        -- merged attributes in merged_met and vertex_properties.
        merge_vertex_attributes(merged_met, vertex_properties, v, name(co), A, BASE, B);

        co_graph := copy of co's graph;
        add vertex v to co_graph with associated merged_met and vertex_properties;
        set co's graph to co_graph;

        recycle co_graph;

end add_composite_vertex;
```

Figure 4.39: Algorithm Sketch for **add_composite_vertex**

5. Module: merge_edge_attributes

This procedure recovers latencies and edge properties from original **CHANGE A**, **BASE**, and **CHANGE B** prototype composite operators for use in the post-merge reconstruction of new composite operators during decomposition recovery. It is

necessary to go the original **CHANGE A, BASE,** and **CHANGE B** prototypes to get the edge attributes given that they are lost in the pre-merge flattening process and thus are absent from the flattened merged prototype.

---

procedure merge_edge_attributes(merged_latency: in out millisec;
streams_properties: in out init_map;
source, sink: op_id;
stream_name, co_name: psdl_id;
A, BASE, B: psdl_program);

Input:

merged_latency: used to return the merged latency
streams_properties: used to return the properties of the edge's data stream
source: op_id identifier for edge's source operator
sink: op_id identifier for edge's sink operator
stream_name: psdl_id name for edge's data stream
co_name: psdl_id name of composite operator to retrieve edge from in A, BASE, B
A: pre-merge version of Change A prototype
BASE: pre-merge version of BASE prototype
B: pre-merge version of Change B prototype

Output:

merged_latency: merged latency for the edge's data stream
streams_properties: properties of the edge's data stream

Figure 4.40: Concrete Interface Specification for **merge_edge_attributes**

```
Algorithm merge_edge_attributes(merged_latency: in out millisec;
                                streams_properties: in out init_map;
                                source, sink: op_id;
                                stream_name, co_name: psdl_id;
                                A, BASE, B: psdl_program);

        a_graph, base_graph, b_graph: psdl_graph;
        a_latency, base_latency, b_latency: millisec := undefined_time;
begin
        initialize a_graph, base_graph, b_graph to empy;

        if co_name is an operator in A then
                a_graph := copy of co_name's graph from A;
                if the edge source, sink, stream_name is in a_graph then
                        a_latency := latency for the edge from a_graph;
                endif;
        endif;

        if co_name is an operator in BASE then
                base_graph := copy of co_name's graph from BASE;
                if the edge source, sink, stream_name is in base_graph then
                        base_latency := latency for the edge from base_graph;
                endif;
        endif;

        if co_name is an operator in B then
                b_graph := copy of co_name's graph from B;
                if the edge source, sink, stream_name is in b_graph then
                        b_latency := latency for the edge from b_graph;
                endif;
        endif;

-- Now, merge the recovered latencies
        if base_latency = a_latency then
                if base_latency = b_latency then
                        merged_latency := base_latency;
                else
                        merged_latency := b_latency;
                endif;
        else
                if base_latency = b_latency then
                        merged_latency := a_latency;
                else
                        if a_latency = b_latency then
                                merged_latency := a_latency;
                        else
                                merged_latency := undefined_time; -- different
                        endif;
                endif;
        endif;
```

Figure 4.41: Algorithm Sketch for **merge_edge_attributes**

```
-- Now, based on which prototype the latency was recovered from, get
-- the corresponding edge_property init_map.

if merged_latency = base_latency and the edge source, sink, stream_name is in base_graph
then
        streams_properties := copy of the edge's properties from base_graph;
elsif merged_latency = a_latency and the edge source, sink, stream_name is in a_graph
then
        streams_properties := copy of the edge's properties from a_graph;
elsif merged_latency = b_latency and the edge source, sink, stream_name is in b_graph
then
        streams_properties := copy of the edge's properties from b_graph;
else
        streams_properties := empty_init_map;
endif;

recycle a_graph, base_graph, b_graph;

end merge_edge_attributes;
```

Figure 4.42: Algorithm Sketch for **merge_edge_attributes** (cont.)

6. Module: merge_composite_elements

This module is used to update new composite operator's states, axioms, informal description, and implementation descriptions by attempting a merge of original composite operators from the **BASE**, **CHANGE A**, and **CHANGE B psdl_program'**s.

procedure merge_composite_elements(A, BASE, B: in psdl_program;
op: in out composite_operator);

Input:

op: composite operator whose elements will be merged
A: pre-merge version of Change A prototype
BASE: pre-merge version of BASE prototype
B: pre-merge version of Change B prototype

Output:

op: composite operator with merged elements

Figure 4.43: Concrete Interface Specification for **merge_composite_elements**

```
Algorithm merge_composite_elements(A, BASE, B: in psdl_program;
                                              co: in out composite_operator);

        co_A, co_BASE, co_B: composite_operator;
        merged_states: type_declaration;
        merged_init: init_map;

begin
        -- first get the composite operators from the original decomposition's.
        -- If one doesn't exist, make a dummy so we can reuse existing functions and
        -- procedures.

        If co is an operator in prototype A then
                co_A := co's definition from A;
        else
                co_A := make_composite_operator(name(co));
        endif;

        If co is an operator in prototype BASE then
                co_BASE := co's definition from BASE;
        else
                co_BASE := make_composite_operator(name(co));
        endif;

        If co is an operator in prototype A then
                co_B := co's definition from B;
        else
                co_B := make_composite_operator(name(co));
        endif;

        -- merge the informal descriptions and assign the merged result to co.
        set_informal_description(merge_text(informal_description(co_BASE),
                                 informal_description(co_A),
                                 informal_description(co_B)), co);

        -- merge the axioms and assign the merged result to co.
        set_axioms(merge_text(axioms(co_BASE), axioms(co_A), axioms(co_B)), co);

        -- merge the implementation descriptions and assign the merged result to co.
        set_implementation_description(merge_text(implementation_description(co_BASE),
                                 implementation_description(co_A),
                                 implementation_description(co_B)), co);

        -- Call merge_states to merge the states and associated values.
        merge_states(merged_states, states(co_BASE), states(co_A), states(co_B),
                                 merged_init, get_init_map(co_BASE), get_init_map(co_A),
                                 get_init_map(co_B));
```

Figure 4.44: Algorithm Sketch for **merge_composite_elements**

```
-- add the states to the new composite operator co.
if merged_states is not empty then
        for each state stream and associated type_name in merged_states
        loop
                add the state stream with associated type_name to composite operator co;
        endloop;
endif;

-- add the initial values for the states to the new composite operator co.
if merged_init  is not empty then
        for each stream and associated initialization expression in merged_init
        loop
                add the stream and associated initialization expression to composite operator co;
        endloop;
endif;

recycle local psdl data structures;

end merge_composite_elements;
```

Figure 4.45: Algorithm Sketch for **merge_composite_elements** (cont.)

7. Module: set_op_id_operation_name

To access many of an operators specification and implementation elements, a variable of type op_id containing the operator's psdl_id is needed. This procedure returns such a variable initialized to the psdl_id name of an operator.

procedure set_op_id_operation_name(id: psdl_id; op:in out op_id);

Input:

id: psdl_id name to be assigned to op
op: used to return op_id identifier for id

Output:

op: op_id identifier for operator psdl_id name "id"

Figure 4.46: Concrete Interface Specification for **set_op_id_operation_name**

```
Algorithm set_op_id_operation_name(id: psdl_id; op:in out op_id)
is
begin
        op.operation_name := id;
        op.type_name := empty;
end set_op_id_operation_name;
```

Figure 4.47: Algorithm Sketch for **set_op_id_operation_name**

8. Module: update_parents_graph

The input and output edges for composite operators other the root are also edges in the their parent's graph. What **update_parents_graph** does is add a composite operator's input and output edges to its parent's **psdl_graph** edge set. An edge is an input edge for a composite operator if the edge's source is not in the operator's edge set. An edge is an output edge for a composite operator if the edge's sink is not in the operator's edge set. It is necessary to go the original **CHANGE A, BASE,** and **CHANGE B** prototypes to get the edge attributes given that they are lost in the pre-merge flattening process and thus are absent from the flattened merged prototype.

```
procedure update_parents_graph(co: composite_operator;
                               A, BASE, B, NEW_PSDL: psdl_program);
Input:
        co: composite operator whose parent's graph will be updated
        A: pre-merge version of Change A prototype
        BASE: pre-merge version of BASE prototype
        B: pre-merge version of Change B prototype
```

Figure 4.48: Concrete Interface Specification for **update_parents_graph**

```
Algorithm update_parents_graph(co: composite_operator;  A, BASE, B, NEW_PSDL: psdl_program);

        child_graph, parent_graph: psdl_graph;
        source_parent_op_id, sink_parent_op_id: op_id;
        parent_co, parent_op: composite_operator;
        graphs_edges: edge_set; streams_properties: init_map;
        streams_latency: millisec := undefined_time;
begin
        child_graph := copy of co's graph;
        parent_graph := copy of co's parent's graph;
        parent_co := co's parent's operator definition;
        streams_properties := empty_init_map;
        graphs_edges := copy of co's graph edge set;

        for each edge E in graphs_edges
        loop
                if the E's sink is not in child_graph then
                        if the E's sink is not in parent_graph then
                                sink_parent_op_id := op_id identifier of sink operator's parent;
                                source_parent_op_id := op_id identifier of source operator's parent;
                                if the edge NE: source_parent_op_id, sink_parent_op_id,
                                        E's stream_name is not in parent_graph
                                then
                                        Call merge_edge_attributes to merge streams_latency and
                                        streams_properties for the edge NE from parent_co's graphs
                                        in A, BASE, B prototypes;

                                        Add the edge NE and associated merged streams_latency and
                                        streams_properties to parent_graph;
                                endif;
                        endif;
                endif;
```

Figure 4.49: Algorithm Sketch for **update_parents_graph**

```
            if the E's source is not in child_graph then
                  if the E's source is not in parent_graph then
                        sink_parent_op_id := op_id identifier of sink operator's parent;
                        source_parent_op_id := op_id identifier of source operator's parent;
                        if the edge NE: source_parent_op_id, sink_parent_op_id,
                              E's stream_name is not in parent_graph
                        then
                              Call merge_edge_attributes to merge streams_latency and
                              streams_properties for the edge NE from parent_co's graphs
                              in A, BASE, B prototypes;

                              Add the edge NE and associated merged streams_latency and
                              streams_properties to parent_graph;
                        endif;
                  endif;
            endif;
      endloop;

      set parent_co's graph to parent_graph;
      recycle local psdl data structures;

end update_parents_graph;
```

Figure 4.50: Algorithm Sketch for **update_parents_graph** (cont.)

9. Module: update_root_edges

At the root operator level, input and output edges go into or come out of composite operators. As input and output edges of root's child operators are copied to root, the edge may have a source or sink that is not a vertex in root's graph. This indicates that the edge begins (sources) or ends (sinks) in a composite operator in root's graph. What **update_root_edges** does is find such sources and sinks and changes their names to the corresponding composite operator names in root. It is necessary to go the original **CHANGE A**, **BASE**, and **CHANGE B** prototypes to get the edge attributes given that they are lost in the pre-merge flattening process and thus are absent from the flattened merged prototype.

```
procedure update_root_edges(co: in out composite_operator;
                                   A, BASE, B, NEW_PSDL: psdl_program);
Input:
      co: root's composite operator definition
      A: pre-merge version of Change A prototype
      BASE: pre-merge version of BASE prototype
      B: pre-merge version of Change B prototype
Output:
      co: root's updated definition
```

Figure 4.51: Concrete Interface Specification for **update_root_edges**

```
Algorithm update_root_edges(co: in out composite_operator; A, BASE, B, NEW_PSDL: psdl_program);

      parent_op: composite_operator;
      root_graph: psdl_graph;
      graphs_edges: edge_set;
      streams_properties: init_map;
      streams_latency: millisec := undefined_time;
      sink_parent_op_id, source_parent_op_id: op_id;

begin
      root_graph := copy of co's graph;
      streams_properties := empty_init_map;
      graphs_edges := copy of edges from root_graph;

      for each edge E in graphs_edges
      loop
            if E's source is not in root_graph then
                  source_parent_op_id := op_id identifier for source's parent;
                  if the edge NE: source_parent_op_id, E's sink, E's stream_name is not in
                  root_graph
                  then
                        Call merge_edge_attributes to merge streams_latency and
                        streams_properties for the edge NE from root's graphs
                        in A, BASE, B prototypes;

                        remove edge E from root_graph;

                        Add the edge NE and associated merged streams_latency and
                        streams_properties to root_graph;
                  end if;
            end if;
```

Figure 4.52: Algorithm Sketch for **update_root_edges**

```
        if E's sink is not in root_graph then
                sink_parent_op_id := op_id identifier for sink's parent;
                if the edge NE: E's source, sink_parent_op_id, E's stream_name is not in
                root_graph
                then
                        Call merge_edge_attributes to merge streams_latency and
                        streams_properties for the edge NE from root's graphs
                        in A, BASE, B prototypes;

                        remove edge E from root_graph;

                        Add the edge NE and associated merged streams_latency and
                        streams_properties to root_graph;
                end if;
        end if;
end loop;

set root's grsph to root_grsph;

recycle local psdl data structures;

end update_root_edges;
```

Figure 4.53: Algorithm Sketch for **update_root_edges** (cont.)

10. Module: set_external_inputs_n_outputs

For composite operators other than the root operator, this procedure labels the source for input edges and the sink for output edges as EXTERNAL:

input streams:
EXTERNAL -> input stream_name -> local sink operator;

output streams:
local source operator -> output stream_name -> EXTERNAL.

It is necessary to go the original **CHANGE A**, **BASE**, and **CHANGE B** prototypes to get the edge attributes given that they are lost in the pre-merge flattening process and thus are absent from the flattened merged prototype.

```
procedure set_external_inputs_n_outputs(co: in out composite_operator;
                                        A, BASE, B, NEW_PSDL: psdl_program);
Input:
        co: composite operator whose graph will be updated
        A: pre-merge version of Change A prototype
        BASE: pre-merge version of BASE prototype
        B: pre-merge version of Change B prototype

Output:
        co: operator with update graph
```

Figure 4.54: Concrete Interface Specification for **set_external_inputs_n_outputs**

```
Algorithm set_external_inputs_n_outputs(co: in out composite_operator;
                                        A, BASE, B, NEW_PSDL: psdl_program);

        parent_op_id: op_id;
        parent_op: composite_operator;
        new_graph, parent_graph: psdl_graph;
        input_streams, output_streams: type_declaration;
        graphs_edges: edge_set;
        streams_properties: init_map;
        streams_latency: millisec := undefined_time;
        external: op_id;
begin
        new_graph := a copy of co's graph;
        input_streams := co's input type_declarations;
        output_streams := co's output type_declarations;
        streams_properties := empty_init_map;
        external := operation_name set to "EXTERNAL"
        graphs_edges := co's graph's edge set;

        -- for inputs, the sink will be local and the source will be EXTERNAL;
        for each stream_name S and associated type_name in input_streams
        loop
                for each edge E in graphs_edges
                loop
                        if S's stream_name = E's stream_name and E's source is not in new_graph
                        then
                                if edge NE: external, E's sink, E's stream_name is not in new_graph
                                then
                                        Call merge_edge_attributes to merge streams_latency and
                                        streams_properties for the edge NE from co's graphs
                                        in A, BASE, B prototypes;

                                        Remove edge E from new_graph;
```

Figure 4.55: Algorithm Sketch for **set_external_inputs_n_outputs**

```
                                        Add the edge NE and associated merged streams_latency and
                                        streams_properties to new_graph;
                                else -- remove redundant stream for the stream_name with e.sink
                                        Remove edge E from new_graph;
                                endif;
                        endif;
                endloop;

        endloop;

        -- for outputs, the sink will be EXTERNAL and the source will be local
        for each stream_name S and associated type_name in output_streams
        loop
                for each edge E in graphs_edges
                loop
                        if S's stream_name = E's stream_name and E's sink is not in new_graph
                        then
                                if edge NE: E's source, external, E's stream_name is not in new_graph
                                then
                                        Call merge_edge_attributes to merge streams_latency and
                                        streams_properties for the edge NE from co's graphs
                                        in A, BASE, B prototypes;

                                        Remove edge E from new_graph;

                                        Add the edge NE and associated merged streams_latency and
                                        streams_properties to new_graph;
                                else -- remove redundant stream for the stream_name with e.sink
                                        remove edge E from new_graph;
                                endif;
                        endif;
                endloop;

        endloop;

        set co's graph to new_graph

        recycle local psdl data structures;

end set_external_inputs_n_outputs;
```

Figure 4.56: Algorithm Sketch for **set_external_inputs_n_outputs** (cont.)

11. Module: copy_streams

This module is used to copy data streams from one composite operator to another. In the context of decomposition recovery, the copy is from the merged prototype's root operator to a composite operator in reconstructed prototype.

```
procedure copy_streams(from_op: composite_operator;
                       to_op: in out composite_operator);

Input:
        from_op: operator that data streams will be copied from.
        to_op: operator that data streams will be copied to.

Output:
        to_op: operator with updated data streams.
```

Figure 4.57: Concrete Interface Specification for **copy_streams**

```
Algorithm copy_streams(from_op: composite_operator;
                  to_op: in out composite_operator);

        to_graph: psdl_graph;
        data_streams: type_declaration;
        to_graph_edges: edge_set;
begin
        to_graph := copy of to_op's graph;
        data_streams := copy of from_op's data streams;
        to_graph_edges := copy of to_graph's edge set;

        for edge E in to_graph_edges
        loop
                for each stream_name S and type_name T in data_streams
                loop
                        if S = E's stream_name then
                                if S's stream_name is not a member of to_op's data streams
                                        and S's stream_name is not in to_ops inputs
                                        and S's stream_name is not in to_ops outputs then
                                                add S, T to to_op's data steams;
                                endif;
                        endif;
                endloop;
        endloop;

        recycle local psdl data structures;
end copy_streams;
```

Figure 4.58: Algorithm Sketch for **copy_streams**

12. Module: finish_composite_operator_construction

By the time this module is called in decomposition recovery processing, a skeletal decomposition structure has been constructed from merged ancestor chains – all operators are in correct structural context with regard to parent-child relationship. However, the composite operator's specification and implementation parts are largely incomplete. What **finish_composite_operator_construction** does is recurse through this skeletal structure filling in the missing specification and implementation parts for these operators.

---

procedure finish_composite_operator_construction(gr: psdl_graph;
A, BASE, B, NEW_PSDL: psdl_program;
co, new_root_co; merged_root_co: psdl_component);

Input:

gr: the composite operator incomplete graph.
A: pre-merge version of Change A prototype.
BASE: pre-merge version of BASE prototype.
B: pre-merge version of Change B prototype.
NEW_PSDL: partially reconstructed prototype.
co: the composite operator to finish reconstructing.
new_root_co: the root operator for the prototype under reconstruction
merged_root_co: the root operator definition from the merged flattened prototype.

Figure 4.59: Concrete Interface Specification for **finish_composite_operator_construction**

Algorithm finish_composite_operator_construction(gr: psdl_graph;
A, BASE, B, NEW_PSDL: psdl_program;
co, new_root_co; merged_root_co: psdl_component);

graphs_vertices: op_id_set;
graphs_edges : edge_set;,
source_not_in_vertices, sink_not_in_vertices: Boolean := True;
local_co: psdl_component;
copy_of_graph: psdl_graph;
merged_type_name: type_name := null_type;

Figure 4.60: Algorithm Sketch for **finish_composite_operator_construction**

```
begin
        graphs_vertices := gr's vertice op_id_set;

        -- recurse down through composite operator graphs setting input and output  stream attributes for
        -- composite operators. When this loop exits, any child composite operator for "co" has been
        -- reconstructed and "co's" graph has been updated and can be used to set "input" and "output"
        -- specification attributes

        for each op_id  ID in graphs_vertices
        loop
                local_co := ID's operator definition from NEW_PSDL;
                if co is a composite operator then
                        copy_of_graph := copy of co's graph;
                        finish_composite_operator_construction(copy_of_graph, A, BASE, B,
                                        NEW_PSDL, local_co, new_root_co, merges_root_co);
                endif;
        endloop;

-- if  there is a source or sink for an edge and the source or sink is not in  the vertices set for the graph, then
-- the edge is an input stream or output stream ; so, assign the stream as an input stream or output stream
-- for the  operator

        local_co := co;

        if local_co is not equal to new_root_co then
                graphs_edges := copy of gr's edge set;

                for each edge E in graphs_edges) loop
                        source_not_in_vertices := True;
                        sink_not_in_vertices := True;

                        for each op_id ID in graphs_vertices loop
                                if E's source = ID then
                                        source_not_in_vertices := False;
                                endif;
                                if E's sink = ID then
                                        sink_not_in_vertices := False;
                                endif;
                        endloop;

                        if source_not_in_vertices then
                                if E's source name is not "EXTERNAL"
                                then
                                        if E's stream_name is not in local_co's inputs
                                        then
                                                Call merge_input_stream_type_names
                                                to get merged_type_name for E from local_co's
                                                definitions in A, BASE, B;
```

Figure 4.61: Algorithm Sketch for **finish_composite_operator_construction** (cont.)

```
                                        Add E's stream_name, merged_type_name,
                                        to local_co's inputs;
                              endif;
                    endif;
          endif;

          if sink_not_in_vertices then
                    if E's sink name is not "EXTERNAL"
                    then
                              if E's stream_name is not in local_co's outputs
                              then
                                        Call merge_output_stream_type_names
                                        to get merged_type_name for E from local_co's
                                        definitions in A, BASE, B;

                                        Add E's stream_name, merged_type_name,
                                        to local_co's outputs;
                              endif;
                    endif;
          endif;
     endloop;
endif;


-- copy over data streams from merged co corresponding to edges
-- in local_co's graph
copy_streams(merged_root_co, local_co);

if local_co is not equal to new_root_co then
     update_parents_graph(local_co, A, BASE, B, NEW_PSDL);
     set_external_inputs_n_outputs(local_co, A, BASE, B, NEW_PSDL);
else
     update_root_edges(new_root_co, A, BASE, B, NEW_PSDL);
endif;

-- set_visible_timers(local_co);

-- merge axioms, implementation descriptions, informal descriptions,
-- and states
merge_composite_elements(A, BASE, B, local_co);

recycle local psdl data structures;

end finish_composite_operator_construction;
```

Figure 4.62: Algorithm Sketch for **finish_composite_operator_construction** (cont.)

13. Module: copy_timing_constraints

This module is used to copy operator's timing constraints (period, finished-within, minimum-calling-period, maximum response time) from one composite operator to another. In the context of decomposition recovery, the copy is from the merged prototype's root operator to a composite operator in the reconstructed prototype.

---

procedure copy_timing_constraints(operator_id: op_id; from_op: composite_operator;
to_op: in out composite_operator);

Input:

operator_id: op_id identifier for composite operator.
from_op: composite operator that values will be copied from.
to_op: composite operator that values will bw copied to.

Output:

to_op: composite operator updated with from_op's timing constraints

---

Figure 4.63: Concrete Interface Specification for **copy_timing_constraints**

Algorithm copy_timing_constraints(operator_id: op_id; from_op: composite_operator;
to_op: in out composite_operator);

begin

Copy operator_id's "period" value from from_op to to_op;

Copy operator_id's "finish_within" value from from_op to to_op;

Copy operator_id's "minimum_calling_period" value from from_op to to_op;

Copy operator_id's "maximum_response_time" value from from_op to to_op;

end **copy_timing_constraints**;

Figure 4.64: Algorithm Sketch for **copy_timing_constraints**

14. Module: copy_exception_triggers

This module is used to copy operator's exception triggers from one composite operator to another. In the context of decomposition recovery, the copy is from the merged prototype's root operator to a composite operator in the reconstructed prototype.

```
procedure copy_exception_triggers(operator_id: op_id;
                                  from_op: composite_operator;
                                  to_op: in out composite_operator);

Input:
        operator_id: op_id identifier for composite operator.
        from_op: composite operator that values will be copied from.
        to_op: composite operator that values will bw copied to.
Output:
        to_op: composite operator updated with from_op's exception triggers.
```

Figure 4.65: Concrete Interface Specification for **copy_exception_triggers**

```
Algorithm copy_exception_triggers(operator_id: op_id;
    from_op: composite_operator; to_op: in out composite_operator);

        local_op_id: op_id := operator_id;
        excep_trigs: excep_trigger_map;
begin
        excep_trigs := copy of from_op's exception_trigger_map;

        for each excep_id  EX and associated expression EXP in excep_trigs loop
                if EX's op_id identifier = operator_id then
                        Copy EX and associated EXP from from_op to to_op;
                endif;
        endloop;
        recycle excep_trigs;

end copy_exception_triggers;
```

Figure 4.66: Algorithm Sketch for **copy_exception_triggers**

15. Module: copy_control_constraints

This module is used to copy operator's control constraints (triggers, execution guards, output guards, and exception triggers) from one composite operator to another. In the context of decomposition recovery, the copy is from the merged prototype's root operator to a composite operator in the reconstructed prototype.

---

procedure copy_control_constraints(operator_id: op_id; gr: psdl_graph;
from_op: composite_operator;
to_op: in out composite_operator);

Input:

operator_id: op_id identifier for composite operator.
gr: copy of from_op's graph.
from_op: composite operator that values will be copied from.
to_op: composite operator being copied to.

Output:

to_op: composite operator updated with from_op's control constraints.

Figure 4.67: Concrete Interface Specification for **copy_control_constraints**

```
Algorithm copy_control_constraints(operator_id: op_id; gr: psdl_graph;
        from_op: composite_operator; to_op: in out composite_operator);

begin
        Copy operator_id's triggers from from_op to to_op;

        Copy operator_id's execution guards from from_op to to_op;

        -- copy output guards from from_op to to_op.
        for each edge E in gr's edge set loop
                if E's source = local_op_id then
                        Copy from_op's output_guard for E's stream_name to to_op;
                endif;
        endloop.

        Copy operator_id's exception triggers from from_op to to_op;

end copy_control_constraints;
```

Figure 4.68: Algorithm Sketch for **copy_control_constraints**

16. Module: copy_vertex_n_edges

This module is used to a copy vertex and corresponding edges from one operator's **psdl_graph** to another. In the context of decomposition recovery, the copy is from the merged prototype's root operator to a composite operator in the prototype under reconstruction.

---

procedure copy_vertex_n_edges(op: op_id; from_graph: psdl_graph; to_graph: in out psdl_graph);

Input:

op: op_id identifier for composite operator.
from_graph: graph thta values will be copied from.
to_graph: graph that values will be copied to.

Output:

to_graph: graph updated with vertex and related edges..

Figure 4.69: Concrete Interface Specification for **copy_vertex_n_edges**

Algorithm copy_vertex_n_edges(op: op_id; from_graph: psdl_graph; to_graph: in out psdl_graph);

```
        local_op_id: op_id := op;
        from_graph_edges, to_graph_edges: edge_set;
begin
        Copy vertex "op" and associated MET and vertex properties from from_graph to to_graph;

        from_graph_edges := copy of from_graph's edges;
        to_graph_edges:= copy of to_graph's edges;

        for each edge E in from_graph_edges loop
                if E's source = local_op_id or E's sink = local_op_id then
                        if E is not in to_graph_edges then
                                copy E and E's latency and edge properties from from_graph
                                to to_graph;
                        endif;
                endif;
        endloop;

        recycle from_graph_edges, to_graph_edges.

end copy_vertex_n_edges;
```

Figure 4.70: Algorithm Sketch for **copy_vertex_n_edges**

17. Modules Taken from [Ref. 2]

Refer to [Ref. 2] for detail.

Module merge_types

Purpose:

used to merge the type_name's of data streams and state streams.

Concrete Interface Specification:

function merge_types(t_base, t_a, t_b: type_name) return type_name;

Module merge_text

Purpose:

used to merge axioms and informal descriptions for composite operators.

Concrete Interface Specification:

function merge_text(BASE, A, B: text) return text;

Module merge_states

Purpose:

used to merge composite operator states and corrsponding initial values.

Concrete Interface Specification:

```
procedure merge_states( MERGE: in out type_declaration;
                        BASE, A, B: in type_declaration;
                        MERGEINIT: in out init_map;
                        BASEINIT, AINIT, BINIT: in init_map);
```

Note: this module has been altered as follows: in [Ref. 2] the cases where the state is only in A or only in B is not accounted for. This module was altered to account for theses cases.

Figure 4.71: Modules Taken from [Ref. 2]

V. IMPLEMENTATION AND TEST

A. IMPLEMENTATION

The Decomposition Recovery Extension to the CAPS Change-Merge Tool is implemented in Ada 95 with the GNAT 3.09 compiler. See Appendix A for source listings.

This implementation (as well as the design) make extensive use of the PSDL Abstract Data Type developed by the CAPS Research Team. Some minor extension were made to this type to accommodate this implementation. These extensions are detailed in Appendix B.

B. TEST

Testing demonstrated correct behavior of ancestor chain recovery and merge on a number of actual PDSL prototypes of various sizes (none which could be considered large), as well as various combinations of ancestor chains developed specifically for test. Conflict reporting and correct automatic conflict resolution for ancestor chain merge were demonstrated as well. See Appendix C for representative test-cases.

Testing also demonstrated correct reconstruction of PSDL prototype decomposition structure from the set of recovered ancestor chains. Correct reconstruction was demonstrated in both the case of conflicting and conflict-free ancestor chain merges.

Time did not permit rigorous analysis of the implementation's performance. However, simple observation suggests that performance is non-linear (but not excessively non-linear) in terms of the number of operators in the prototype.

VI. CONCLUSION

A. WHAT HAS BEEN DONE AND WHY IT IS IMPORTANT

The purpose of the CAPS Change-Merge Tool is to provide an automated integration capability "...for combining and integrating the contributions of different people working on the same prototype" [Ref. 2]. The current Change-Merge Tool provides an automated, reliable, fast integration capability but loses the decomposition structure of the prototype in the integration process. The decomposition structure of a PSDL prototype is the critical design information which provides understandability for designers. Even for small PSDL prototypes, the lack of decomposition structure in a merged prototype makes it very difficult to continue prototyping efforts using the merged prototype as the basis. Manual recovery of decomposition structure is simply too time consuming. Thus, to have other than limited practical value in a rapid prototyping environment, the CAPS Change-Merge Tool must automatically recover decomposition structure as part of the merge process.

What has been accomplished in this thesis is the software design and Ada implementation of an extension to the Change-Merge Tool which provides a capability to do just that – automatically recover design decomposition structure for merged PSDL prototypes. The merge and automatic conflict identification and resolution algorithms of this extension are based in the formal theory developed in [Ref. 1]. Thus, it has a degree of reliability based on a formalized approach. As for recovered design, merge of non-overlapping structural changes produces a decomposition structure which exactly reflects structural changes to a prototype. Merge of overlapping or conflicting structural changes produces a decomposition structure which closely approximates structural changes to a prototype and provides a very reasonable design decomposition structure from which post-merge prototyping can continue.

Thus, with the Decomposition Recovery Extension, the CAPS Change-Merge Tool developed in [Ref. 2] not only provides a fast, automated, reliable integration capability for integrating PSDL prototypes, it now provides a design decomposition structure for merged prototypes as well. Thus, the post-merge delay incurred by loss of decomposition structure is eliminated.

B. WHAT STILL NEEDS TO BE DONE

With regards to the CAPS Change-Merge capability in general, some of what still needs to be done is given in [Ref. 2].

With regards to the Decomposition Recovery Extension, a number of things still need to be accomplished. The extension still needs to be integrated with the current Change-Merge Tool. This will at least mean code changes to the Change-Merge Tool to integrate the most recent version of PSDL_TYPE and save un-expanded versions of CHANGE A, BASE, and CHANGE B prototypes. Actual integration of the Decomposition Recovery Extension is provided through a single call to procedure **decompose_graph_pkg.decompose_graph**.

The prototype flattening process which proceeds prototype merge destroys all composite operators except root. The **reconstruct_prototype** function has to recreate many of these composite operators during prototype reconstruction. In some cases, it goes to un-expanded versions of the pre-merge prototypes to retrieve composite operator elements and then merges these elements to derive the corresponding element for the new composite operator. It may be the case that composite operator reconstruction could be largely accomplished by merging the versions of the original operators in the un-expanded pre-merge prototypes. Much of the source code of [Ref. 2] could be reused to in such an effort.

In a few cases, software to recover some of the elements of composite operator specification and implementation parts is not yet in place. These elements include *keywords*, *visible timers*, *exceptions*, and *specified maximum execution times.*

Also, test of prototype reconstruction has been limited to smaller sized PSDL prototypes. Thus, as larger prototypes become available, they could be used as test cases.

Ancestor chain merge conflict reporting could be improved to provide more detail. Currently, only one of possibly many conflicts is reported, and this only with a general statement that a conflict has occurred accompanied by a display of conflict terms. The user must determine where the conflict occurred by inspecting the displayed conflict terms. See Figure 6.1 for detail of a merge conflict report.

ONE OR MORE CONFLICTS IN ANCESTOR CHAIN RECOVERY FOR: atomic_op

```
<root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7>
[<root_op->op_1->op_2->op_3->op_4->op_5->op_6>]
<root_op->op_1->op_2->op_3->op_8->op_4> =
<root_op->op_1->op_2->op_3->op_8->op_4> U
<root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7>=
(***conflict***) =
<root_op->op_1->op_2->op_3(op_8->op_4 U op_4->op_5->op_6->op_7)>
```

Figure 6.1: Example ancestor chain merge conflict report

APPENDIX A. ADA IMPLEMENTATION

This appendix gives the source listings for the Ada packages which make up the Decomposition Recovery Extension to the CAPS Change-Merge Tool. The specification and body is given for each package.

1. decompose_graph_pkg

```
with text_io; use text_io;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_graph_pkg; use psdl_graph_pkg;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_io; use psdl_io;
with extended_ancestor_pkg; use extended_ancestor_pkg;
with ancestor_chains_pkg; use ancestor_chains_pkg;
with reconstruct_prototype_utilities_pkg; use reconstruct_prototype_utilities_pkg;
package decompose_graph_pkg is

-- find ancestor chain calls recursive function 'recover_chain' to
-- recover 'N's ancestor chain from a prototype's decomposition structure.
-- The recovered chain sequence will be of the form:
--
--                  [root_id]
--
--                          or
--
--                  [root_id, 0 or more chain elements, 'N's immediate ancestor],
--
-- where a chain element is a psdl_id name for an operator. In the first
-- form, 'N's immediate ancestor is root.
--
        function find_ancestor_chain(N, root_id: psdl_id; P: psdl_program)
        return extended_ancestor;



--
-- Apply the merge formula
--     A[BASE]B = [A pseudo-difference BASE]
--                     union
--                [A intersection B]
--                     union
--                [B pseudo-difference BASE]
--
-- to 'N's recovered ancestor chains from prototypes A, BASE, and B.
--
-- If the result of the union operation is a null_component, then
-- a merging conflict has occured
```

```
--
        procedure merge_ancestor_chains(A_CHAIN, BASE_CHAIN, B_CHAIN:
                extended_ancestor; MERGE_CHAIN: in out extended_ancestor);


-- For each improper ancestor, calculate the greatest lower
-- bound in the extended ancestor lattice of the conflicting
-- chains and assign it as the proper ancestor chain for
-- atomic operator 'N'.
--
        procedure  resolve_conflicts(ea_map: in out ancestor_chains;
                                                    root_op: psdl_id);

-- Reconstruct the decomposition structure for the merged prototype
-- from the merged ancestor chains
--
        function reconstruct_prototype(MERGE, A, BASE, B: psdl_program;
                        ANCESTORS: ancestor_chains)
        return psdl_program;

-- procedure decompose_graph is the interface to the PSDL prototype
-- decomposition recovery sub-system. The first 3 psdl_program arguments
-- are the pre-expanded versions of PSDL prototypes for change A, the BASE,
-- and change B. MERGE is the flattened result of the merge of A, BASE,
-- and B. THE PSDL prototype with recovered decomposition structure is
-- returned in NEW_PSDL.
--
        procedure decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE: psdl_program;
              NEW_PSDL: in out psdl_program);

end decompose_graph_pkg;

package body decompose_graph_pkg is

-- find ancestor chain calls recursive function 'recover_chain' to
-- recover 'N's ancestor chain from a prototype's decomposition structure.
-- The recovered chain sequence will be of the form:
--
--              [root_id]
--
--                      or
--
--              [root_id, 0 or more chain elements, 'N's immediate ancestor],
--
-- where a chain element is a psdl_id name for an operator. In the first
-- form, 'N's immediate ancestor is root.
--
function find_ancestor_chain(N, root_id: psdl_id; P: psdl_program)
return extended_ancestor
is

        ancestor: extended_ancestor := null_ancestor;
        ancestor_id: psdl_id;
```

```
-- recursive function that constructs ancestor chain
function recover_chain(ancestor: extended_ancestor; operator_id,
        root_id: psdl_id; P: psdl_program)
return psdl_id
is

        ancestor_id: psdl_id;
begin
        -- if have reached the root operator, unwind the recursion
        if eq(operator_id, root_id) then
                return root_id;
        else -- recurse to get next ancestor
                ancestor_id := recover_chain(ancestor,
                                        get_ancestor(operator_id, P), root_id, P);
                -- recursion unwinding, append operator_id's ancestor to chain
                append_ancestor(ancestor, ancestor_id);
                return operator_id;
        end if;
end recover_chain;

begin -- find_ancestor_chain

        ancestor := build_proper_ancestor(empty);

        -- make sure we don't try to find the root operator's chain; the root
        -- operator is the composite operator in the MERGED psdl_program where
        -- 'N' is the key. The root operator will be an element of every 'N's
        -- ancestor chain
        if not eq(N, root_id) and member(N, P) then
                -- recursively construct N's ancestor chain
                ancestor_id := recover_chain(ancestor,
                                        get_ancestor(N, P), root_id, P);

                -- append N's immediate ancestor to the chain
                append_ancestor(ancestor, ancestor_id);
        end if;
        return ancestor;

end find_ancestor_chain;



--
-- Apply the merge formula
--     A[BASE]B =  [A pseudo-difference BASE]
--                      union
--                 [A intersection B]
--                      union
--                 [B pseudo-difference BASE]
--
-- to 'N's recovered ancestor chains from prototypes A, BASE, and B.
--
-- If the result of the union operation is a null_component, then
-- a merging conflict has occured
```

```
--
procedure merge_ancestor_chains(A_CHAIN, BASE_CHAIN, B_CHAIN:
extended_ancestor; MERGE_CHAIN: in out extended_ancestor)
is
    a_pseudodiff_base,
    a_intersection_b,
    b_pseudodiff_base,
    union_term: extended_ancestor := null_ancestor;
begin
    -- first try the simple cases
    if eq(A_CHAIN, BASE_CHAIN) then
        MERGE_CHAIN:= build_proper_ancestor(get_chain(B_CHAIN));
    elsif eq(B_CHAIN, BASE_CHAIN) then
        MERGE_CHAIN:= build_proper_ancestor(get_chain(A_CHAIN));
    elsif eq(A_CHAIN, B_CHAIN) then
        MERGE_CHAIN:= build_proper_ancestor(get_chain(B_CHAIN));
    else -- have to apply the merge formula
        a_pseudodiff_base := pseudo_difference(A_CHAIN, BASE_CHAIN);
        a_intersection_b := intersection(A_CHAIN, B_CHAIN);
        b_pseudodiff_base := pseudo_difference(B_CHAIN, BASE_CHAIN);

        -- combine the three merge formula terms in two union operations
        union_term := union(a_pseudodiff_base, a_intersection_b);
        MERGE_CHAIN := union(union_term, b_pseudodiff_base);
        if MERGE_CHAIN = null_ancestor then -- conflict
            MERGE_CHAIN := build_improper_ancestor(get_chain(A_CHAIN),
                                        get_chain(BASE_CHAIN),
                                        get_chain(B_CHAIN));
        end if;
        recycle_extended_ancestor(a_pseudodiff_base);
        recycle_extended_ancestor(a_intersection_b);
        recycle_extended_ancestor(b_pseudodiff_base);
        recycle_extended_ancestor(union_term);
    end if;
end merge_ancestor_chains;


-- For each improper ancestor, calculate the greatest lower
-- bound in the extended ancestor lattice of the conflicting
-- chains and assign it as the proper ancestor chain for
-- atomic operator 'N'.
procedure  resolve_conflicts(ea_map: in out ancestor_chains; root_op: psdl_id)
is
    scan_ea_map: ancestor_chains;
        ea_1: extended_ancestor;
begin
    ancestor_chains_map_inst_pkg.assign(scan_ea_map, ea_map);
    for N: psdl_id, ea: extended_ancestor in
                ancestor_chains_map_inst_pkg.scan(scan_ea_map)
    loop
        if type_of_ancestor(ea) = improper then
            ancestor_chains_map_inst_pkg.remove(N, ea_map);
            ancestor_chains_map_inst_pkg.bind(N,
                                    resolve_conflict(ea), ea_map);
```

```
                else
                        if eq(ea, empty_extended_ancestor) then
                        -- account for the pathelogical case where N is in
                        -- the merged graph but merging the ancestor chains
                        -- from A, BASE, and B resulted in an empty chain
                                ea_1 := build_proper_ancestor(empty);
                                append_ancestor(ea_1, root_op);
                ancestor_chains_map_inst_pkg.remove(N, ea_map);
                ancestor_chains_map_inst_pkg.bind(N, ea_1, ea_map);
                end if;
            end if;
      end loop;
          ancestor_chains_map_inst_pkg.recycle(scan_ea_map);
end resolve_conflicts;

-- For each improper ancestor, output infomative conflict message
--
procedure  report_conflicts(ea_map: ancestor_chains)
is
begin
      for N: psdl_id, ea: extended_ancestor in
                  ancestor_chains_map_inst_pkg.scan(ea_map)
      loop
            if type_of_ancestor(ea) = improper then
                        put_conflict_message(N, ea);
                else
                        if eq(ea, empty_extended_ancestor) then
                        -- account for the pathelogical case where N is in
                        -- the merged graph but merging the ancestor chains
                        -- from A, BASE, and B resulted in an empty chain
                                put(convert(N));
                                put_line(" HAS EMPTY MERGED CHAIN, POSSIBLE MERGE
                                                CONFLICT");
                                put_line("ASSIGNING ROOT OPERATOR AS PARENT");
                end if;
            end if;
      end loop;
end report_conflicts;


-- Reconstruct the decomposition structure for the merged prototype
-- from the merged ancestor chains
function reconstruct  prototype(MERGE, A, BASE, B: psdl_program;
                                             ANCESTORS: ancestor_chains)
return psdl_program
is

          NEW_PSDL: psdl_program;
          co_node, ancestor_node, new_root_op, merges_root_op: composite_operator;
          new_atomic_op, atomic_op: atomic_operator;
          root_id: psdl_id;
          chain: psdl_id_sequence;
          merges_graph, ancestor_graph, root_op_graph: psdl_graph;
```

```
        root_op_id, op, atomic_op_id: op_id;

begin
        -- put_line("reconstruct_prototype_called");

        assign(NEW_PSDL, empty_psdl_program);

        root_id := find_root(MERGE);
        set_op_id_operation_name(root_id, root_op_id);
        merges_root_op := fetch(MERGE, root_id);
        assign(merges_graph, graph(merges_root_op));
        new_root_op := make_composite_operator(root_id);

        -- bind NEW_PSDL's root operator
        bind(root_id, new_root_op, NEW_PSDL);

        -- for every atomic operator in the extended_ancestor map loop
        for atomic_id: psdl_id, ea: extended_ancestor in
                                ancestor_chains_map_inst_pkg.scan(ANCESTORS)
        loop
                -- get ancestor_id's merged ancestor chain; note that every
                -- chain will at least have the root operator (root_id) as
                -- an element
                assign(chain, get_chain(ea));

                -- for every chain_element in atomic_id's ancestor chain starting
                -- with the root element, construct the composite component
                -- decomposition structure corresponding to the sequence of
                -- ancestor chain elements
                for chain_element: psdl_id in psdl_id_sequence_pkg.scan(chain) loop
                        -- if the composite operator corresponding to the chain
                        -- element already exists in NEW_PSDL, then get it; note
                        -- that NEW_PSDL's root operator has already been created
                        if member(chain_element, NEW_PSDL) then
                                co_node := fetch(NEW_PSDL, chain_element);
                        else
                                -- create a new composite operator for chain_element

                                co_node := make_composite_operator(chain_element);

                                -- make co_node's parent operator ancestor_node
                                set_parent(co_node, ancestor_node);

                                -- add co_node to ancestor_node's implementation
                                -- graph vertex set.
                                -- First, initialize op_id op=>operator_name to
                                -- chain_element name
                                set_op_id_operation_name(chain_element, op);

                                -- have an op_id, now add vertex for composite
                                -- operator to the parent graph; try to retrieve
                                -- vertext attributes from A, BASE, B entries
                                -- fro the composite
```

```
                add_composite_vertex(op,
                                ancestor_node, A, BASE, B);

                -- bind composite co_node to NEW_PSDL;
                bind(chain_element, co_node, NEW_PSDL);
        end if;
        -- co_node becomes ancestor_node for next loop iteration
        -- or for atomic_id when loop exits
        ancestor_node:= co_node;
end loop;
recycle(chain);

-- At this point, the decomposition structure correspoinding to
-- atomic_id's ancestor chain is in place. The next step is to
-- copy atomic_id's attributes from the big root merged
-- composite operator to atomic_id's parent composite operator.

-- get the atomic psdl_component corresponding to atomic_id
-- from MERGE
atomic_op := fetch(MERGE, atomic_id);

-- create a new operator from the operator fetched from MERGE
new_atomic_op := make_atomic_operator(psdl_name => name(atomic_op),
                        ada_name => ada_name(atomic_op),
                        gen_par => generic_parameters(atomic_op),
                        keywords => keywords(atomic_op),
                        informal_description =>
                                informal_description(atomic_op),
                        axioms => axioms(atomic_op),
                        input => inputs(atomic_op),
                        output => outputs(atomic_op),
                        state => states(atomic_op),
                        initialization_map => get_init_map(atomic_op),
                        exceptions => exceptions(atomic_op),
                        specified_met =>
                        specified_maximum_execution_time(atomic_op));

-- atomic_op's parent => ancestor_node
set_parent(new_atomic_op, ancestor_node);

-- Create an op_id corresponding to atomic_id for use in copying
-- atomic_id's edges, timers, timing and control constraints
-- from the big root merged composite to atomic_id
-- parent composite's implementation part
set_op_id_operation_name(atomic_id, atomic_op_id);

-- update parent's graph - copy over any edges from the
-- big root merged composite to atomic_op_id parent
-- graph for which atomic_op_id is ether a source or a
-- sink.
-- First, get a copy of the atomic operator's parent's
-- graph
assign(ancestor_graph, graph(ancestor_node));
```

```
            -- copy the vertex and edges from merges_graph to ancestor_graph
            copy_vertex_n_edges(atomic_op_id, merges_graph, ancestor_graph);

            -- assign the updated graph to ancestor_node
            set_graph(ancestor_graph, ancestor_node);
            recycle(ancestor_graph);

            -- update parent's timer ops - copy over any timer
            -- operations corresponding to atomic_op_id from the
            -- big root merged composite to atomic_op_id's parent
            copy_timer_operations(atomic_op_id, merges_root_op,
                                                        ancestor_node);

            -- update parent's output guards, exception triggers, execution
            -- guards, and triggers - copy over any control constraints
            -- corresponding to atomic_op_id from the big root
            -- merged composite to atomic_op_id's parent
            copy_control_constraints(atomic_op_id, merges_graph,
                                          merges_root_op, ancestor_node);

            -- update parent's periods, finished withins, minimum calling
            -- periods, and maximum response times - copy over any timing
            -- constraints corresponding to atomic_op_id from the big
            -- root merged composite to atomic_op_id's parent
            copy_timing_constraints(atomic_op_id, merges_root_op,
                                                        ancestor_node);

            -- bind new atomic operator to NEW_PSDL;
            bind(atomic_id, new_atomic_op, NEW_PSDL);


      end loop;
      recycle(merges_graph);

-- At this point, a skeletal decomposition structure is in place - all of the
-- composite operators are in place with partially completed specification
-- and implementation portions.
--
-- Next, finish up construction of the each composite operator in NEW_PSDL;
-- input edges, output edges, state edges, smet's, exceptions, initial states,
-- and other attributes will have to be set in each composite operator's
-- specification and implementation part.
-- Starting with the root operator, recurse through composite operator graphs to
-- finish reconstruction of each composite operator's specification and
-- implementation parts
      -- put(NEW_PSDL);
      assign(root_op_graph, graph(new_root_op));
      finish_composite_operator_construction(graph(new_root_op), A, BASE, B,
                                                   NEW_PSDL, new_root_op,
                                                   new_root_op, merges_root_op);
      recycle(root_op_graph);

      -- put_line("leaving reconstruct_prototype");
      return NEW_PSDL;
```

```
end reconstruct_prototype;

-- procedure decompose_graph is the interface to the PSDL prototype
-- decomposition recovery sub-system. The first 3 psdl_program arguments
-- are the pre-expanded versions of PSDL prototypes for change A, the BASE,
-- and change B. MERGE is the flattened result of the merge of A, BASE,
-- and B. THE PSDL prototype with recovered decomposition structure is
-- returned in NEW_PSDL.
--
procedure decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE: psdl_program;
                          NEW_PSDL: in out psdl_program)
is
     root_op: psdl_id;
         ancestors: ancestor_chains;
         MERGE_CHAIN, A_CHAIN, BASE_CHAIN, B_CHAIN:
                         extended_ancestor := null_ancestor;
begin
         ancestor_chains_map_inst_pkg.assign(ancestors, empty_ancestor_chains);
     assign(NEW_PSDL, empty_psdl_program);

     -- need the root operator for find_ancestor_chain.
     root_op := find_root(MERGE);
         -- put_line(convert(root_op));

     for id: psdl_id, c : psdl_component in psdl_program_map_pkg.scan(MERGE) loop
                 if component_category(c) = psdl_operator then
                         if component_granularity(c) = atomic then
                         A_CHAIN := find_ancestor_chain(id, root_op, A_PSDL);
                         BASE_CHAIN := find_ancestor_chain(id, root_op, BASE_PSDL);
                         B_CHAIN := find_ancestor_chain(id, root_op, B_PSDL);
                         merge_ancestor_chains(A_CHAIN, BASE_CHAIN, B_CHAIN,
                                                                  MERGE_CHAIN);
                                 ancestor_chains_map_inst_pkg.bind(id, MERGE_CHAIN, ancestors);
                         end if;
                 end if;
     end loop;

         report_conflicts(ancestors);
         resolve_conflicts(ancestors, root_op);
         put_ancestor_chains(ancestors);

         assign(NEW_PSDL, reconstruct_prototype(MERGE, A_PSDL, BASE_PSDL,
                                                          B_PSDL, ancestors));
         ancestor_chains_map_inst_pkg.recycle(ancestors);

end decompose_graph;

end decompose_graph_pkg;
```

2. extended_ancestor_pkg

```
with text_io;                  use text_io;
with psdl_graph_pkg;           use psdl_graph_pkg;
with psdl_program_pkg;         use psdl_program_pkg;
with psdl_component_pkg;       use psdl_component_pkg;
with psdl_concrete_type_pkg;   use psdl_concrete_type_pkg;
Package extended_ancestor_pkg is

        -- Discriminant for type extended_ancestor_record
        type ancestor_type is (proper, improper);

        -- storage for both "proper" and "improper" ancestor chains.
        type extended_ancestor_record
                (ancestor: ancestor_type)
        is private;

        type extended_ancestor is access extended_ancestor_record;

-- "proper" ancestor is an element of the set of all finite sequences partially
-- ordered by the prefix ordering [2]. in this implementation, proper_ancestor is
-- a pointer to an extended_ancestor_record with discrimant "ancestor => proper".
-- This subtype is used to store an atomic operator's properly formed ancestor
-- chain as a sequence of psdl_id names of composite operators.

        subtype proper_ancestor is extended_ancestor(ancestor => proper);

-- "improper" ancestor is an improper data element representing a least upper
-- bound for a set of incomparable "proper" elements in the extended ancestor
-- lattice. used to represent merging conflicts [2] ]. in this implementation,
-- improper_ancestor is a pointer to an extended_ancestor_record with discrimant
-- "ancestor => improper". This subtype is used to store conflicting
-- proper ancestor chains for subsequent conflict reporting and resolution.

        subtype improper_ancestor is extended_ancestor(ancestor =>improper);

        null_ancestor: constant extended_ancestor := null;

        empty_extended_ancestor: extended_ancestor;

        -- raised when null_ancestor is unexpectedly encountered.
        undefined_ancestor: exception;

        -- raised when an undefined ancestor chainis unexpectedly encountered.
        undefined_ancestor_chain: exception;

        -- raised when comparison of an improper-to-proper ancestor is unexpectedly
        -- attempted
        ancestor_type_mismatch: exception;

        -- returns an extended_ancestor's discriminant: "proper" or "improper"
        function type_of_ancestor(ea: extended_ancestor) return ancestor_type;
```

```
-- returns a proper ancestor with an empty ancestor chain
function empty_ancestor return proper_ancestor;

-- appends a component's psdl_id name to an ancestor chain
-- procedure append_ancestor(ea: in out extended_ancestor; ancestor_id: psdl_id);
procedure append_ancestor(ea: extended_ancestor; ancestor_id: psdl_id);

-- assigns the ancestor chain of one proper_ancestor to another; recycles the
-- assignee's existing ancestor chain prior to new assignment
procedure assign_chain(ea_1: in out proper_ancestor; ea_2: proper_ancestor);

-- assigns an ancestor chain to a proper_ancestor; recycles the assignee's
-- existing ancestor chain prior to new assignment
procedure assign_chain(ea: in out proper_ancestor; chain: psdl_id_sequence);

-- returns a proper ancestor initialized to the supplied ancestor chain sequence
function build_proper_ancestor(ea_chain: psdl_id_sequence)
return proper_ancestor;

-- returns an improper ancestor initialized to the supplied ancestor chain
-- sequences
function build_improper_ancestor(a_chain, base_chain, b_chain:
                                 psdl_id_sequence) return improper_ancestor;

-- determine where the merge conflicts occurred and output an informative
-- message detailing the conflict in reasonable depth. (This is a first cut just
-- to have something in place. The plan is to revisit once more general things
-- are accomplished.)
procedure put_conflict_message(N: psdl_id; ia: improper_ancestor);


-- determine where the merge conflicts occurred, resolve the conflict, and
-- return the resolved ancestor chain as s proper_ancestor
--
function resolve_conflict(ia: improper_ancestor) return proper_ancestor;

-- return True if the first chain argument is a prefix of the second; False otherwise
function is_prefix_of(ea_1, ea_2: extended_ancestor) return boolean;

-- return True if the first chain argument is a prefix of the second; False otherwise
function is_prefix_of(chain_1, chain_2: psdl_id_sequence) return boolean;

-- determines equality for both proper_ancestor's and improper_ancestor's
function eq(ea_1, ea_2: extended_ancestor) return boolean;

-- intersection operation for extended_ancestor; returns the result in a newly
-- allocated extended_ancestor_record
function intersection(ea_1, ea_2: extended_ancestor) return extended_ancestor;

-- intersection operation for ancestor chains (psdl_id_sequence); returns the result
-- in a new psdl_id_sequence
function intersection(chain_1, chain_2: psdl_id_sequence) return psdl_id_sequence;
```

```
-- The union operation for extended_ancestor; return a new
-- extended_ancestor if the union was successful; otherwise, return
-- null extended_ancestor.
function union(ea_1, ea_2: extended_ancestor) return extended_ancestor;

-- The union operation for ancestor chain sequences; return a new
-- psdl_id_sequence ancestor chain if the union was successful; otherwise, return
-- conflict = True if the union could not be formed.
procedure union(chain_1, chain_2: psdl_id_sequence; result: in out
psdl_id_sequence; conflict: in out boolean);

-- The Brouwerian Algebra pseudo-difference operation as defined on the
-- Extended Ancestor Lattice
function pseudo_difference(ea_1, ea_2: extended_ancestor) return extended_ancestor;

-- The Brouwerian Algebra pseudo-difference operation as defined on the
-- Extended Ancestor Lattice
function pseudo_difference(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence;

-- returns the greatest common prefix of 2 ancestor chain sequences
function greatest_common_prefix(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence;

-- recycle storage for proper or improper extended_ancestor_records
procedure recycle_extended_ancestor(ea: in out extended_ancestor);

-- get the psdl_id identifier of  a component's ancestor
function get_ancestor(id: psdl_id; p: psdl_program) return psdl_id;

-- return, as a proper ancestor, the greatest lower bound (least common prefix) for
-- the conflicting chains of the improper
function get_greatest_lower_bound(ea: improper_ancestor)
return proper_ancestor;

-- return a proper ancestor's psdl_id_sequence ancestor chain
function get_chain(ea: extended_ancestor) return psdl_id_sequence;

procedure put_chain( chain: psdl_id_sequence; add_cr: Boolean);

procedure put_ancestor(ea: extended_ancestor);

private

type extended_ancestor_record(ancestor: ancestor_type)
is record
        case ancestor is
                when proper =>
                        chain: psdl_id_sequence;
                when improper =>
                        chain_A: psdl_id_sequence;
                        chain_BASE: psdl_id_sequence;
                        chain_B: psdl_id_sequence;
                end case;
```

```
        end record;

end extended_ancestor_pkg;


package body extended_ancestor_pkg
is

-- returns an extended_ancestor's descriminant: "proper" or "improper"
function type_of_ancestor(ea: extended_ancestor) return ancestor_type
is
begin
        if ea = null_ancestor then raise undefined_ancestor; end if;
        return ea.ancestor;
end type_of_ancestor;

-- returns a proper ancestor with and empty ancestor chain
function empty_ancestor return proper_ancestor
is
        result: proper_ancestor;
begin
        -- result := build_proper_ancestor(psdl_id_seq_inst_pkg.empty);
        result := build_proper_ancestor(empty);
        return result;
end empty_ancestor;

-- appends an ancestor to an extended_ancestor's ancestor chain
-- procedure append_ancestor(ea: in out extended_ancestor; ancestor_id: psdl_id)
procedure append_ancestor(ea: extended_ancestor; ancestor_id: psdl_id)
is
begin
        if ea = null_ancestor then raise undefined_ancestor; end if;
        assign(ea.chain, add(ancestor_id, ea.chain));
end append_ancestor;

-- assigns the ancestor chain of one proper_ancestor to another; recycles the
-- assignee's existing ancestor chain prior to new assignment
procedure assign_chain(ea_1: in out proper_ancestor; ea_2: proper_ancestor)
is
begin
        if ea_1 = null_ancestor or ea_2 = null_ancestor then
                raise undefined_ancestor;
        end if;
        assign(ea_1.chain, ea_2.chain);
end assign_chain;

-- assigns an ancestor chain to a proper_ancestor; recycles the assignee's
-- existing ancestor chain prior to new assignment
procedure assign_chain(ea: in out proper_ancestor; chain: psdl_id_sequence)
is
begin
        if ea = null_ancestor then raise undefined_ancestor; end if;
        assign(ea.chain, chain);
end assign_chain;
```

```
-- returns a proper ancestor initialized to the supplied ancestor chain sequence
function build_proper_ancestor(ea_chain: psdl_id_sequence)
return proper_ancestor
is
        pa: proper_ancestor;
begin
        pa := new extended_ancestor_record(ancestor => proper);
        assign(pa.chain, ea_chain);
        return pa;
end build_proper_ancestor;

-- returns an improper ancestor initialized to the supplied ancestor chain
-- sequences
function build_improper_ancestor(a_chain, base_chain, b_chain: psdl_id_sequence)
return improper_ancestor
is
        ia: improper_ancestor;
begin
        ia := new extended_ancestor_record(ancestor => improper);
        assign(ia.chain_A, a_chain);
        assign(ia.chain_BASE, base_chain);
        assign(ia.chain_B, b_chain);
return ia;
end build_improper_ancestor;

function is_prefix_of(ea_1, ea_2: extended_ancestor) return boolean
is
begin
        return (is_prefix_of(ea_1.chain, ea_2.chain));

end is_prefix_of;
-- return True if the first chain argument is a prefix of the second; False otherwise
function is_prefix_of(chain_1, chain_2: psdl_id_sequence)
return boolean
is
        length_chain_1: natural;
        length_chain_2: natural;
        chain_2_prefix: psdl_id_sequence;
        is_prefix: Boolean := False;
begin

        -- empty chain is prefix of all chains
        if equal(chain_1, empty) then return True; end if;

        length_chain_1:= length(chain_1);
        length_chain_2:= length(chain_2);
        -- first, check the lengths of the chains
        if length_chain_1 > length_chain_2 then -- can't be prefix of shorter chain
                is_prefix := False;
        else
                assign(chain_2_prefix, empty);
                fetch(chain_2, 1, length_chain_1, chain_2_prefix);
                -- put_line("is_prefix_of CHAINS");
```

```
                -- put_chain(chain_2_prefix, True);
                -- put_chain(chain_1, True);

                if equal(chain_1, chain_2_prefix) then
                        is_prefix := True;
                else
                        is_prefix := False;
                end if;
                recycle(chain_2_prefix);
        end if;

        return is_prefix;

end is_prefix_of;

-- determines equality for both proper_ancestor's and improper_ancestor's
function eq(ea_1, ea_2: extended_ancestor)
return boolean
is
begin
        if ea_1 = null_ancestor or ea_2 = null_ancestor then
                raise undefined_ancestor;
        end if;
        if type_of_ancestor(ea_1) = type_of_ancestor(ea_2) then
                if type_of_ancestor(ea_1) = proper then
                        if equal(ea_1.chain, ea_2.chain) then
                                return True;
                        else
                                return False;
                        end if;
                else -- improper ancestor
                        if equal(ea_1.chain_A, ea_2.chain_A)
                        and equal(ea_1.chain_BASE, ea_2.chain_BASE)
                        and equal(ea_1.chain_B, ea_2.chain_B) then
                                return True;
                        else
                                return False;
                        end if;
                end if;
        else
                raise ancestor_type_mismatch;
        end if;
end eq;

-- recycle storage for proper or improper extended_ancestor_records
procedure recycle_extended_ancestor(ea: in out extended_ancestor)
is
begin
        if ea = null_ancestor then raise undefined_ancestor; end if;
        if type_of_ancestor(ea) = proper then
                recycle(ea.chain);
        else
                recycle(ea.chain_A);
                recycle(ea.chain_BASE);
```

```
                recycle(ea.chain_B);
        end if;
        ea := null_ancestor;
end recycle_extended_ancestor;

-- get the psdl_id identifier of  a component's ancestor
function get_ancestor(id: psdl_id; p: psdl_program)
return psdl_id
is
begin
        return (name(parent(fetch(p, id))));
end get_ancestor;

-- return a proper ancestor's psdl_id_sequence ancestor chain
function get_chain(ea: extended_ancestor)
return psdl_id_sequence
is
begin
        if ea = null_ancestor then raise undefined_ancestor; end if;
        if type_of_ancestor(ea) = proper then
                return ea.chain;
        else
                raise ancestor_type_mismatch;
        end if;

end get_chain;

procedure put_chain( chain: psdl_id_sequence; add_cr: Boolean)
is
        low, high: natural;
begin
        low := 1;
        high := length(chain);
        if high > 0 then
                for i in low .. high
                loop
                        put(convert(fetch(chain, i)));
                        if i /= high then put("->"); end if;
                end loop;
        else
                put("EMPTY CHAIN");
        end if;
        -- add a carriage return
        if add_cr then put_line(""); end if;
end put_chain;

procedure put_ancestor(ea: extended_ancestor)
is
begin
        if ea = null_ancestor then raise undefined_ancestor; end if;

        if type_of_ancestor(ea) = proper then
                put_chain(ea.chain, True);
        else
```

```
                put_line("***IMPROPER ANCESTOR***");
                put("A:    ");
                put_chain(ea.chain_A, True);
                put("BASE: ");
                put_chain(ea.chain_BASE, True);
                put("B:    ");
                put_chain(ea.chain_B, True);
        end if;
end put_ancestor;


-- returns the greatest common prefix of 2 ancestor chain sequences
function greatest_common_prefix(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence
is
        length_chain_1: natural := length(chain_1);
        length_chain_2: natural := length(chain_2);
        compare_limit: natural;
        result: psdl_id_sequence;
        I: natural := 1;
        elements_match: Boolean := True;
begin
        assign(result, empty);
        -- first, set the range for chain element comparison
        if length_chain_1 > length_chain_2 then
                compare_limit := length_chain_2;
        else
                compare_limit := length_chain_1;
        end if;

        -- extract the greatest common prefix and store it in "result"
        while I <= compare_limit and elements_match loop
                if eq(fetch(chain_1, I), fetch(chain_2, I)) then
                        assign(result, add(fetch(chain_1, I), result));
                        I := I + 1;
                else
                        elements_match := False;
                end if;
        end loop;
        return result;
end greatest_common_prefix;


-- intersection operation for extended_ancestor; returns the result in a newly
-- allocated extended_ancestor_record; greatest lower bounds are set intersections for
-- extended ancestor lattice
function intersection(ea_1, ea_2: extended_ancestor) return extended_ancestor
is
        result: extended_ancestor := null_ancestor;
begin
        if ea_1 = null_ancestor or ea_2 = null_ancestor then
                raise undefined_ancestor;
        end if;
```

```
        if is_prefix_of(get_chain(ea_1), get_chain(ea_2)) then
                -- put_line("is_prefix_of(get_chain(ea_1), get_chain(ea_2)");
                result := build_proper_ancestor(get_chain(ea_1));
        elsif is_prefix_of(get_chain(ea_2), get_chain(ea_1)) then
                result := build_proper_ancestor(get_chain(ea_2));
        else
                result := build_proper_ancestor(
                        greatest_common_prefix(get_chain(ea_2), get_chain(ea_1)));
        end if;
        return result;
end intersection;


-- The Brouwerian Algebra pseudo-difference operation as defined on the
-- Extended Ancestor Lattice
function pseudo_difference(ea_1, ea_2: extended_ancestor) return extended_ancestor
is
    result: extended_ancestor := null_ancestor;
begin
    if ea_1 = null_ancestor or ea_2 = null_ancestor then
        raise undefined_ancestor;
    end if;

    if is_prefix_of(ea_1, ea_2) then
        result := build_proper_ancestor(empty);
    else
        result := build_proper_ancestor(get_chain(ea_1));
    end if;
    return result;
end pseudo_difference;


-- The union operation for extended_ancestor; return a new
-- extended_ancestor if the union was successful; otherwise, return
-- null extended_ancestor.
function union(ea_1, ea_2: extended_ancestor) return extended_ancestor
is
    result: extended_ancestor := null_ancestor;
begin
    if ea_1 = null_ancestor or ea_2 = null_ancestor then
        raise undefined_ancestor;
    end if;

    if eq(ea_1, empty_extended_ancestor) then
        result := build_proper_ancestor(get_chain(ea_2));
    elsif eq(ea_2, empty_extended_ancestor) then
        result := build_proper_ancestor(get_chain(ea_1));
    elsif is_prefix_of(ea_1, ea_2) then
        result := build_proper_ancestor(get_chain(ea_2));
    elsif is_prefix_of(ea_2, ea_1) then
        result := build_proper_ancestor(get_chain(ea_1));
    else -- conflict; indicate by returning null_ancestor
        result := null_ancestor;
    end if;
```

```
    return result;
end union;


-- return, as a proper ancestor, the greatest lower bound (greatest common prefix) for
-- the conflicting chains of the improper
function get_greatest_lower_bound(ea: improper_ancestor)
return proper_ancestor
is
    result: proper_ancestor;
    chain_1, chain_2: psdl_id_sequence;
begin
    if ea = null_ancestor then raise undefined_ancestor; end if;

    assign(chain_1, empty);
    assign(chain_2, empty);
    assign(chain_1, greatest_common_prefix(ea.chain_A, ea.chain_BASE));
    assign(chain_2, greatest_common_prefix(chain_1, ea.chain_B));

    result := build_proper_ancestor(chain_2);

    recycle(chain_1);
    recycle(chain_2);

    return result;
end get_greatest_lower_bound;



-- intersection operation for ancestor chains (psdl_id_sequence); returns the result
-- in a new psdl_id_sequence
function intersection(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence
is
        result: psdl_id_sequence;
begin
        assign(result, empty);
        if is_prefix_of(chain_1, chain_2) then
                assign(result, chain_1);
        elsif is_prefix_of(chain_2, chain_1) then
                assign(result, chain_2);
        else
                assign(result, greatest_common_prefix(chain_2, chain_1));
        end if;
        return result;
end intersection;


-- The union operation for ancestor chain sequences; return a new
-- psdl_id_sequence ancestor chain if the union was successful; otherwise, return
-- conflict = True if the union could not be formed.
procedure union(chain_1, chain_2: psdl_id_sequence;
                              result: in out psdl_id_sequence;
                              conflict: in out boolean)
```

```
is
begin
        conflict := False;
        if is_prefix_of(chain_1, chain_2) then
                assign(result, chain_2);
        elsif is_prefix_of(chain_2, chain_1) then
                assign(result, chain_1);
        else -- conflict; indicate by returning conflict = True
                conflict := True;
        end if;
end union;


-- The Brouwerian Algebra pseudo-difference operation as defined on the
-- Extended Ancestor Lattice
function pseudo_difference(chain_1, chain_2: psdl_id_sequence)
return psdl_id_sequence
is
        result: psdl_id_sequence;
begin
        assign(result, empty);
        if not is_prefix_of(chain_1, chain_2) then
                assign(result, chain_1);
        end if;
        return result;
end pseudo_difference;


procedure put_conflict_message(N: psdl_id; ia: improper_ancestor)
is
        lcp, term_1, union_term, union_term_imp,
        a_pseudodiff_base, a_intersection_b,
        b_pseudodiff_base, b_pseudodiff_base_imp: psdl_id_sequence;

        lcp_len: natural := 0;

        conflict: Boolean := False;
begin
        if ia = null_ancestor then raise undefined_ancestor; end if;

        assign(a_pseudodiff_base, empty);
        assign(a_intersection_b, empty);
        assign(b_pseudodiff_base, empty);
        assign(union_term, empty);
        assign(b_pseudodiff_base_imp, empty);
        assign(union_term_imp, empty);
        assign(lcp, empty);

        -- reconstruct the 3 terms from the conflicting merge
        assign(a_pseudodiff_base, pseudo_difference(ia.chain_A, ia.chain_BASE));
        assign(a_intersection_b, intersection(ia.chain_A, ia.chain_B));
        assign(b_pseudodiff_base, pseudo_difference(ia.chain_B, ia.chain_BASE));

        -- output the common part of the conflict message
```

```
        put("ONE OR MORE CONFLICTS IN ANCESTOR CHAIN RECOVERY FOR: ");
        put_line(convert(N));
        put("<"); put_chain(ia.chain_A, False); put_line(">");
        put("[<"); put_chain(ia.chain_BASE, False);  put_line(">]");
        put("<"); put_chain(ia.chain_b, False); put_line("> =");

        union(a_pseudodiff_base, a_intersection_b, union_term, conflict);

        -- find the proper elements of the 2 conflicting terms

        assign(lcp, greatest_common_prefix(union_term, b_pseudodiff_base));
        -- find the improper elements of the 2 confliction terms
        lcp_len := length(lcp);
        fetch(b_pseudodiff_base, lcp_len+1, length(b_pseudodiff_base),
                        b_pseudodiff_base_imp);
        fetch(union_term, lcp_len+1, length(union_term),
                union_term_imp);

        put("<"); put_chain(b_pseudodiff_base, False);
        put(">"); put_line(" U ");
        put("<"); put_chain(union_term, False);
        put_line(">=");
        put_line("(***conflict***) = ");
        put("<"); put_chain(lcp, False);
        put("("); put_chain(b_pseudodiff_base_imp, False);
        put(" U ");
        put_chain(union_term_imp, False); put_line(")>");
        put_line("");

        recycle (a_pseudodiff_base);
        recycle (a_intersection_b);
        recycle (b_pseudodiff_base);
        recycle (union_term);
        recycle (b_pseudodiff_base_imp);
        recycle (union_term_imp);
        recycle (lcp);
end put_conflict_message;



-- determine where the merge conflicts occurred, resolve the conflict, and
-- return the resolved ancestor chain as s proper_ancestor
--
function resolve_conflict(ia: improper_ancestor) return proper_ancestor
is
        lcp, union_term, a_pseudodiff_base,
        a_intersection_b, b_pseudodiff_base: psdl_id_sequence;

        conflict: Boolean := False;

        resolved_chain: proper_ancestor;
begin
        if ia = null_ancestor then raise undefined_ancestor; end if;
```

```
        assign(a_pseudodiff_base, empty);
        assign(a_intersection_b, empty);
        assign(b_pseudodiff_base, empty);
        assign(union_term, empty);
        assign(lcp, empty);

        -- reconstruct the 3 terms from the conflicting merge
        assign(a_pseudodiff_base, pseudo_difference(ia.chain_A, ia.chain_BASE));
        assign(a_intersection_b, intersection(ia.chain_A, ia.chain_B));
        assign(b_pseudodiff_base, pseudo_difference(ia.chain_B, ia.chain_BASE));

        -- reapply the union operation to determine which terms conflict, and try to
        -- reduce to two conflict terms given that the union operation is commutative.
        -- first, try to reduce the terms of first union operation of the merge.
        union(a_pseudodiff_base, a_intersection_b, union_term, conflict);

        -- find the proper elements of the 2 conflicting terms
        -- union_term = a_pseudodiff_base U a_intersection_b
        assign(lcp, greatest_common_prefix(union_term, b_pseudodiff_base));

        resolved_chain := build_proper_ancestor(lcp);

        recycle (a_pseudodiff_base);
        recycle (a_intersection_b);
        recycle (b_pseudodiff_base);
        recycle (union_term);
        recycle (lcp);

        return resolved_chain;

end resolve_conflict;


begin
        empty_extended_ancestor := build_proper_ancestor(empty);
end extended_ancestor_pkg;
```

3. reconstruct_prototype_utilities_pkg

```
with text_io; use text_io;
with System;
with expression_pkg; use expression_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_graph_pkg; use psdl_graph_pkg;
with psdl_program_pkg; use psdl_program_pkg;
package reconstruct_prototype_utilities_pkg is

-- create a composite vertex and add it to co's graph. The vertex
-- attributes are merged from the corresponding attributes in
-- the un-expanded prototypes A, BASE, and B.
--
procedure add_composite_vertex(v: op_id; co: in out composite_operator;
                                         A, BASE, B: psdl_program);

-- update composite operator's states, axioms, informal description, and
-- implementation descriptions by attempting a merge of original
-- composite operators from the BASE, CHANGE A, and CHANGE B psdl_programs.
procedure merge_composite_elements(A, BASE, B: in psdl_program;
                                   co: in out composite_operator);

-- recurse through composite operator graphs to finish reconstruction of composite
-- operators' specification and implementation graphs
procedure finish_composite_operator_construction(gr: psdl_graph;
                  A, BASE, B, NEW_PSDL: psdl_program;
                  co, new_root_co, merged_root_co: psdl_component);

-- copy operator's timing constraints (period, fw, mcp, mrt) from one composite operator
-- to another
procedure copy_timing_constraints(operator_id: op_id; from_op: composite_operator;
                                  to_op: in out composite_operator);

-- copy operator's control constraints (triggers, execution guards, output guards, and
-- exception triggers) from one composite operator to another
procedure copy_control_constraints(operator_id: op_id; gr: psdl_graph;
        from_op: composite_operator; to_op: in out composite_operator);

-- copy operator and corresponding edges from one psdl_graph operator to another
procedure copy_vertex_n_edges(op: op_id; from_graph: psdl_graph; to_graph: in out psdl_graph);

-- set the op_id argument's operation_name field to the psdl_id argument
procedure set_op_id_operation_name(id: psdl_id; op:in out op_id);

procedure copy_timer_operations(op: op_id; to_node: in out composite_operator;
                                from_node: composite_operator);

end reconstruct_prototype_utilities_pkg;

package body reconstruct_prototype_utilities_pkg is
```

```
-- Taken from Dampier's dissertation
function merge_types(t_base, t_a, t_b: type_name) return type_name
is
begin
        if equal(t_base, t_a)
        then
                if equal(t_base, t_b)
                then
                        return(t_base);
                else
                        return(t_b);
                end if;
        else
                if equal(t_base, t_b)
                then
                        return(t_a);
                else
                        if equal(t_a, t_b)
                        then
                                return(t_a);
                        else
                                return null_type;
                        end if;
                end if;
        end if;

end merge_types;

-- This procedure recovers output stream type names
-- from composite operators from origanal A, BASE, and B
-- prototypes for use in the post-merge -- reconstruction
-- of composite operators during decomposition recovery.
-- It is necessary to go the original A, BASE, and B
-- prototypes to get the output stream type names given that they
-- are lost in the pre-merge flattening process and thus are
-- absent from the flattened merged prototype.
--
procedure merge_output_stream_type_names(merged_type_name: in out type_name;
                                        id, stream_name: psdl_id;
                                        A, BASE, B: psdl_program)
is
        a_name, base_name, b_name: type_name := null_type;
        op: composite_operator;
begin

        if member(id, A) then
                op := fetch(A, id);
                if member(stream_name, outputs(op)) then
                        a_name := type_of(stream_name, op);
                end if;
        end if;

        if member(id, BASE) then
                op := fetch(BASE, id);
```

```
                if member(stream_name, outputs(op)) then
                        base_name := type_of(stream_name, op);
                end if;
        end if;

        if member(id, B) then
                op := fetch(B, id);
                if member(stream_name, outputs(op)) then
                        b_name := type_of(stream_name, op);
                end if;
        end if;

        merged_type_name := merge_types(base_name, a_name, b_name);

end merge_output_stream_type_names;

-- This procedure recovers input stream type names
-- from composite operators from origanal A, BASE, and B
-- prototypes for use in the post-merge -- reconstruction
-- of composite operators during decomposition recovery.
-- It is necessary to go the original A, BASE, and B
-- prototypes to get the input stream type names given that they
-- are lost in the pre-merge flattening process and thus are
-- absent from the flattened merged prototype.
--
procedure merge_input_stream_type_names(merged_type_name: in out type_name;
                                        id, stream_name: psdl_id;
                                        A, BASE, B: psdl_program)
is
        a_name, base_name, b_name: type_name := null_type;
        op: composite_operator;
begin

        if member(id, A) then
                op := fetch(A, id);
                if member(stream_name, inputs(op)) then
                        a_name := type_of(stream_name, op);
                end if;
        end if;

        if member(id, BASE) then
                op := fetch(BASE, id);
                if member(stream_name, inputs(op)) then
                        base_name := type_of(stream_name, op);
                end if;
        end if;

        if member(id, B) then
                op := fetch(B, id);
                if member(stream_name, inputs(op)) then
                        b_name := type_of(stream_name, op);
                end if;
        end if;
```

```
          merged_type_name := merge_types(base_name, a_name, b_name);

end merge_input_stream_type_names;

-- This procedure recovers mets and vertex properties
-- from composite operators from origanl A, BASE, and B
-- prototypes for use in the post-merge -- reconstruction
-- of composite operators during decomposition recovery.
-- It is necessary to go the original A, BASE, and B
-- prototypes to get the vertex attributes given that they
-- are lost in the pre-merge flattening process and thus are
-- absent from the flattened merged prototype.
--
procedure merge_vertex_attributes(merged_met: in out millisec;
                                  vertex_properties: in out init_map;
                                  op: op_id; co_name: psdl_id;
                                  A, BASE, B: psdl_program)
is
        a_graph, base_graph, b_graph: psdl_graph;
        a_diff_base, b_diff_base, a_int_b,
        a_met, base_met, b_met: millisec := undefined_time;
begin

        assign(a_graph, empty_psdl_graph);
        assign(base_graph, empty_psdl_graph);
        assign(b_graph, empty_psdl_graph);

        if member(co_name, A) then
                assign(a_graph, graph(fetch(A, co_name)));
                if has_vertex(op, a_graph) then
                        a_met := maximum_execution_time(op, a_graph);
                end if;
        end if;

        if member(co_name, BASE) then
                assign(base_graph, graph(fetch(BASE, co_name)));
                if has_vertex(op, base_graph) then
                        base_met := maximum_execution_time(op, base_graph);
                end if;
        end if;

        if member(co_name, B) then
                assign(b_graph, graph(fetch(B, co_name)));
                if has_vertex(op, b_graph) then
                        b_met := maximum_execution_time(op, b_graph);
                end if;
        end if;


-- Taken from Dampier's dissertation
        if a_met <= b_met then
                a_int_b := b_met;
        else
                a_int_b := a_met;
```

```
end if;

if base_met <= a_met
    -- then a_diff_base := system.max_int;
then
        a_diff_base := undefined_time;
else
        a_diff_base := a_met;
end if;

if base_met <= b_met
-- then b_diff_base := system.max_int;
then
        b_diff_base := undefined_time;
else
        b_diff_base := b_met;
end if;

if a_diff_base <= a_int_b then
        if a_diff_base <= b_diff_base then
                merged_met := a_diff_base;
        else
                merged_met := b_diff_base;
        end if;

else
        if a_int_b <= b_diff_base then
                merged_met := a_int_b;
        else
                merged_met := b_diff_base;
        end if;
end if;


-- Now, based on which prototype the met was recovered from, get
-- the corresponding vertex_property init_map.

if merged_met = base_met and has_vertex(op, base_graph) then
        assign(vertex_properties,
                get_properties(op, base_graph));
elsif merged_met = a_met and has_vertex(op, a_graph) then
        assign(vertex_properties,
                get_properties(op, a_graph));
elsif merged_met = b_met and has_vertex(op, b_graph) then
        assign(vertex_properties,
                get_properties(op, b_graph));
else
        assign(vertex_properties, empty_init_map);
end if;

recycle(a_graph);
recycle(base_graph);
recycle(b_graph);
```

```
end merge_vertex_attributes;

-- create a composite vertex and add it to co's graph. The vertex
-- attributes are merged from the corresponding attributes in
-- the un-expanded prototypes A, BASE, and B.
--
procedure add_composite_vertex(v: op_id; co: in out composite_operator;
                                                    A, BASE, B: psdl_program)
is
        co_graph: psdl_graph;
        op: psdl_component;
        vertex_properties: init_map;
        merged_met: millisec := undefined_time;
begin
        assign(co_graph, graph(co));
        assign(vertex_properties, empty_init_map);

        merge_vertex_attributes(merged_met, vertex_properties, v, name(co),
                                        A, BASE, B);

        set_graph(add_vertex(v, co_graph, merged_met, vertex_properties), co);

        recycle(co_graph);

end add_composite_vertex;


-- This procedure recovers latencies and edge properties
-- from composite operators from origanl A, BASE, and B
-- prototypes for use in the post-merge -- reconstruction
-- of composite operators during decomposition recovery.
-- It is necessary to go the original A, BASE, and B
-- prototypes to get the edge attributes given that they
-- are lost in the pre-merge flattening process and thus are
-- absent from the flattened merged prototype.
--
procedure merge_edge_attributes(merged_latency: in out millisec;
                                streams_properties: in out init_map;
                                source, sink: op_id;
                                stream_name, co_name: psdl_id;
                                A, BASE, B: psdl_program)
is
        a_graph, base_graph, b_graph: psdl_graph;
        a_latency, base_latency, b_latency: millisec := undefined_time;
begin

        assign(a_graph, empty_psdl_graph);
        assign(base_graph, empty_psdl_graph);
        assign(b_graph, empty_psdl_graph);

        if member(co_name, A) then
                assign(a_graph, graph(fetch(A, co_name)));
                if has_edge(source, sink, stream_name, a_graph) then
                        a_latency := latency(source, sink, a_graph);
```

```
            end if;
        end if;

        if member(co_name, BASE) then
            assign(base_graph, graph(fetch(BASE, co_name)));
            if has_edge(source, sink, stream_name, base_graph) then
                base_latency := latency(source, sink, base_graph);
            end if;
        end if;

        if member(co_name, B) then
            assign(b_graph, graph(fetch(B, co_name)));
            if has_edge(source, sink, stream_name, b_graph) then
                b_latency := latency(source, sink, b_graph);
            end if;
        end if;


-- Taken from Dampier's dissertation; system.max_int
-- is returned in the dissertation code if A /= BASE /= B
-- whereas this code returns undefined_time for latency
--
    if base_latency = a_latency then
        if base_latency = b_latency then
            merged_latency := base_latency;
        else
            merged_latency := b_latency;
        end if;
    else
        if base_latency = b_latency then
            merged_latency := a_latency;
        else
            if a_latency = b_latency then
                merged_latency := a_latency;
            else
                merged_latency := undefined_time; -- different
                                                  -- from Dampier
            end if;
        end if;
    end if;


        -- Now, based on which prototype the latency was recovered from, get
        -- the corresponding edge_property init_map.

        if merged_latency = base_latency and
                        has_edge(source, sink, stream_name, base_graph) then
            assign(streams_properties,
                get_properties(source, sink, stream_name,
                                base_graph));
        elsif merged_latency = a_latency and
                        has_edge(source, sink, stream_name, a_graph) then
            assign(streams_properties,
                get_properties(source, sink, stream_name,
```

```
                                        a_graph));
    elsif merged_latency = b_latency and
                          has_edge(source, sink, stream_name, b_graph) then
        assign(streams_properties,
               get_properties(source, sink, stream_name,
                                        b_graph));
    else
        assign(streams_properties, empty_init_map);
    end if;

    recycle(a_graph);
    recycle(base_graph);
    recycle(b_graph);

end merge_edge_attributes;

-- Taken from Dampier's dissertation and used here to merge axioms
-- and informal descriptions for composite operators.
--
function merge_text(BASE, A, B: text) return text
is
begin
    if eq(BASE, empty) and eq(A, empty) and eq(B, empty)
    then
        return empty;
    else
        if eq(BASE, A)
        then
            if not eq(BASE, B)
            then
                return B;
            else
                return BASE;
            end if;
        else
            if eq(BASE, B)
            then
                return A;
            else
                if eq(A, B)
                then
                    return A;
                else
                    return convert("**Text Conflict**");
                end if;
            end if;
        end if;
    end if;

end merge_text;


-- Taken from Dampier's dissertation
procedure merge_states(   MERGE: in out type_declaration;
```

```
                              BASE, A, B: in type_declaration;
                              MERGEINIT: in out init_map;
                              BASEINIT, AINIT, BINIT: in init_map)
is
     init_value: expression;
     base_type, a_type, b_type: type_name;
begin
     assign(MERGE, empty_type_declaration);
     for id: psdl_id, t: type_name in type_declaration_pkg.scan(BASE)
     loop
          if member(id, A) and member(id, B)
          then
               a_type := type_declaration_pkg.fetch(A, id);
               b_type := type_declaration_pkg.fetch(B, id);
               bind(id, merge_types(t, a_type, b_type), MERGE);
               assign(init_value, init_map_pkg.fetch(BASEINIT, id));
               if eq(init_value, init_map_pkg.fetch(AINIT, id))
               then
                    if eq(init_value, init_map_pkg.fetch(BINIT, id))
                    then
                         bind(id, init_value, MERGEINIT);
                    else
                         bind(id, init_map_pkg.fetch(BINIT, id), MERGEINIT);
                    end if;
               else
                    if eq(init_value, init_map_pkg.fetch(BINIT, id))
                    then
                         bind(id, init_map_pkg.fetch(AINIT, id), MERGEINIT);
                    else
                         if eq(init_map_pkg.fetch(AINIT, id),
                                        init_map_pkg.fetch(BINIT, id))
                         then
                              bind(id, init_map_pkg.fetch(AINIT, id),

                                                       MERGEINIT);
                         else
                              bind(id, conflict_expression, MERGEINIT);
                         end if;
                    end if;
               end if;
          end if;
     end loop;
     for id: psdl_id, t: type_name in type_declaration_pkg.scan(A)
     loop
          if not member(id, BASE) and member(id, B)
          then
               base_type := null_type;
               b_type := type_declaration_pkg.fetch(B, id);
               bind(id, merge_types(base_type, t, b_type), MERGE);
               assign(init_value, init_map_pkg.fetch(AINIT, id));
               if eq(init_value, init_map_pkg.fetch(BINIT, id))
               then
                    bind(id, init_value, MERGEINIT);
               else
```

```
                    bind(id, conflict_expression, MERGEINIT);
                end if;
            end if;
            -- if the state is only in A, then add it to the reconstruction;
            -- NOTE: this condition is not accounted for in Dampier's code
            --
            if not member(id, BASE) and not member(id, B)
            then
                bind(id, t, MERGE);
                bind(id, init_map_pkg.fetch(AINIT, id), MERGEINIT);
            end if;
        end loop;

        for id: psdl_id, t: type_name in type_declaration_pkg.scan(B)
        loop
            if not member(id, BASE) and member(id, A)
            then
                base_type := null_type;
                a_type := type_declaration_pkg.fetch(A, id);
                bind(id, merge_types(base_type, a_type, t), MERGE);
                assign(init_value, init_map_pkg.fetch(BINIT, id));
                if eq(init_value, init_map_pkg.fetch(AINIT, id))
                then
                    bind(id, init_value, MERGEINIT);
                else
                    bind(id, conflict_expression, MERGEINIT);
                end if;
            end if;
            -- if the state is only in B, then add it to the reconstruction;
            -- NOTE: this condition is not accounted for in Dampier's code
            --
            if not member(id, BASE) and not member(id, A)
            then
                bind(id, t, MERGE);
                bind(id, init_map_pkg.fetch(BINIT, id), MERGEINIT);
            end if;
        end loop;

end merge_states;

-- Taken from Dampier's dissertation
function merge_id_sets(BASE, A, B: psdl_id_set) return psdl_id_set
is
        A_DIFF_BASE, B_DIFF_BASE, MERGE: psdl_id_set;
begin
        assign(A_DIFF_BASE, empty);
        assign(B_DIFF_BASE, empty);
        assign(MERGE, empty);
        difference(A, BASE, A_DIFF_BASE);
        difference(B, BASE, B_DIFF_BASE);
        for id: psdl_id in psdl_id_set_pkg.scan(A)
        loop
            if member(id, B)
            then
```

```
                add(id, MERGE);
            end if;
        end loop;
        for id: psdl_id in psdl_id_set_pkg.scan(A_DIFF_BASE)
        loop
            if not member(id, MERGE)
            then
                add(id, MERGE);
            end if;
        end loop;
        for id: psdl_id in psdl_id_set_pkg.scan(B_DIFF_BASE)
        loop
            if not member(id, MERGE)
            then
                add(id, MERGE);
            end if;
        end loop;
        return MERGE;
end merge_id_sets;

-- update composite operator's states, axioms, informal description, and
-- implementation descriptions by attempting a merge of original
-- composite operators from the BASE, CHANGE A, and CHANGE B psdl_programs.
procedure merge_composite_elements(A, BASE, B: in psdl_program;
                                   co: in out composite_operator)
is
        co_A, co_BASE, co_B: composite_operator;
        recycle_A, recycle_BASE, recycle_B: Boolean := False;
        merged_states: type_declaration;
        merged_init: init_map;

begin
        -- first get the composite operators from the original decomposition's.
        -- If one doesn't exist, make a dummy so we can reuse existing functions and
        -- procedures.
        if member(name(co), A) then
            co_A := fetch(A, name(co));
        else
            co_A := make_composite_operator(name(co));
            recycle_A := True;
        end if;

        if member(name(co), BASE) then
            co_BASE := fetch(BASE, name(co));
        else
            co_BASE := make_composite_operator(name(co));
            recycle_BASE := True;
        end if;

        if member(name(co), B) then
            co_B := fetch(B, name(co));
        else
            co_B := make_composite_operator(name(co));
            recycle_B := True;
```

```
        end if;

--      Don't have to do
--      assign(op.keyw, merge_id_sets(keywords(co_BASE), keywords(co_A),
--                      keywords(co_B)));

        -- merge the informal descriptions
        set_informal_description(merge_text(informal_description(co_BASE),
              informal_description(co_A),
              informal_description(co_B)), co);

        -- merge the axioms
        set_axioms(merge_text(axioms(co_BASE), axioms(co_A), axioms(co_B)), co);

        -- merge the implementation descriptions
        set_implementation_description(merge_text(implementation_description(co_BASE),
              implementation_description(co_A),
              implementation_description(co_B)), co);

        -- merge the states
        merge_states(merged_states, states(co_BASE), states(co_A), states(co_B),
                     merged_init, get_init_map(co_BASE), get_init_map(co_A),
                     get_init_map(co_B));

        -- add the states to the new composite operator
        if not equal(merged_states, empty_type_declaration) then
                for id: psdl_id, t: type_name in type_declaration_pkg.scan(merged_states)
                loop
                        add_state(id, t, co);
                end loop;
        end if;

        -- add the initial values for the states to the new composite operator
        if not init_map_pkg.equal(merged_init, empty_init_map) then
                for stream: psdl_id, e: expression in init_map_pkg.scan(merged_init)
                loop
                        add_initialization(stream, e, co);
                end loop;
        end if;

        recycle(merged_states);
        recycle(merged_init);

        -- merge the execptions
--      assign(op.execp,  merge_id_sets(co_BASE),
--                              merge_id_sets(co_A),
--                              merge_id_sets(co_B));
--
        if recycle_A then recycle(co_A); end if;
        if recycle_BASE then recycle(co_BASE); end if;
        if recycle_B then recycle(co_B); end if;

end merge_composite_elements;
```

```
-- set the op_id argument's operation_name field to the psdl_id argument
procedure set_op_id_operation_name(id: psdl_id; op:in out op_id)
is
begin
        op.operation_name := id;
        op.type_name := empty;
end set_op_id_operation_name;

-- add the child composite operator's input and output stream edges to
-- its parent's psdl_graph.
procedure update_parents_graph(co: composite_operator;
                                A, BASE, B, NEW_PSDL: psdl_program)
is
        child_graph, parent_graph: psdl_graph;
        source_parent_op_id, sink_parent_op_id: op_id;
        parent_co, parent_op: composite_operator;
        graphs_edges: edge_set;
        streams_properties: init_map;
        streams_latency: millisec := undefined_time;

begin
        assign( child_graph, graph(co));
        assign( parent_graph, graph(parent(co)));
        parent_co := parent(co);
        assign( streams_properties, empty_init_map);

        edge_set_pkg.assign(graphs_edges, edges(child_graph));

        for e: edge in edge_set_pkg.scan(graphs_edges)
        loop
                if not has_vertex(e.sink, child_graph) then
                        if not has_vertex(e.sink, parent_graph) then
                                -- get the sources's parent
                                parent_op := parent(get_definition(NEW_PSDL, e.sink));
                                set_op_id_operation_name(name(parent_op), sink_parent_op_id);
                                parent_op := parent(get_definition(NEW_PSDL, e.source));
                                set_op_id_operation_name(name(parent_op), source_parent_op_id);
                                if not has_edge(source_parent_op_id,
                                                sink_parent_op_id, e.stream_name, parent_graph)
                                then
                                        merge_edge_attributes(streams_latency, streams_properties,
                                                source_parent_op_id,
                                                sink_parent_op_id, e.stream_name, name(parent_co),
                                                A, BASE, B);
                                        assign(parent_graph,
                                                add_edge(source_parent_op_id, sink_parent_op_id,
                                                e.stream_name, parent_graph,
                                                streams_latency,
                                                streams_properties));
                                end if;
                        end if;
                end if;
                if not has_vertex(e.source, child_graph) then
```

```
                    if not has_vertex(e.source, parent_graph) then
                        parent_op := parent(get_definition(NEW_PSDL, e.sink));
                        set_op_id_operation_name(name(parent_op), sink_parent_op_id);
                        parent_op := parent(get_definition(NEW_PSDL, e.source));
                        set_op_id_operation_name(name(parent_op), source_parent_op_id);
                        if not has_edge(source_parent_op_id,
                                sink_parent_op_id, e.stream_name, parent_graph)
                        then
                            merge_edge_attributes(streams_latency, streams_properties,
                                source_parent_op_id,
                                sink_parent_op_id, e.stream_name, name(parent_co),
                                A, BASE, B);
                            assign(parent_graph,
                                add_edge(source_parent_op_id, sink_parent_op_id,
                                e.stream_name, parent_graph,
                                streams_latency,
                                streams_properties));
                        end if;
                    end if;
                end if;
            end loop;

            set_graph(parent_graph, parent_co);

            recycle(streams_properties);
            recycle(child_graph);
            recycle(parent_graph);
            edge_set_pkg.recycle(graphs_edges);

        end update_parents_graph;

        procedure update_root_edges(co: in out composite_operator;
                                    A, BASE, B, NEW_PSDL: psdl_program)
        is
            parent_op: composite_operator;
            root_graph: psdl_graph;
            graphs_edges: edge_set;
            streams_properties: init_map;
            streams_latency: millisec := undefined_time;
            sink_parent_op_id, source_parent_op_id: op_id;

        begin
            assign(root_graph, graph(co));
            assign(streams_properties, empty_init_map);
            edge_set_pkg.assign(graphs_edges, edges(root_graph));

            for e: edge in edge_set_pkg.scan(graphs_edges)
            loop
                if not has_vertex(e.source, root_graph) then
                    parent_op := parent(get_definition(NEW_PSDL, e.source));
                    set_op_id_operation_name(name(parent_op), source_parent_op_id);
                    if not has_edge(source_parent_op_id, e.sink, e.stream_name, root_graph)
                    then
                        merge_edge_attributes(streams_latency, streams_properties,
```

```
                                        source_parent_op_id,
                                        e.sink, e.stream_name, name(co),
                                        A, BASE, B);
                              assign(root_graph, remove_edge(e, root_graph));
                              assign(root_graph,
                                   add_edge(source_parent_op_id, e.sink,
                                   e.stream_name, root_graph,
                                   streams_latency,
                                   streams_properties));
                    end if;
          end if;
          if not has_vertex(e.sink, root_graph) then
                    parent_op := parent(get_definition(NEW_PSDL, e.sink));
                    set_op_id_operation_name(name(parent_op), sink_parent_op_id);
                    if not has_edge(e.source, sink_parent_op_id, e.stream_name, root_graph)
                    then
                              merge_edge_attributes(streams_latency, streams_properties,
                                        e.source,
                                        sink_parent_op_id, e.stream_name, name(co),
                                        A, BASE, B);
                              assign(root_graph, remove_edge(e, root_graph));
                              assign(root_graph,
                                   add_edge(e.source, sink_parent_op_id,
                                   e.stream_name, root_graph,
                                   streams_latency,
                                   streams_properties));
                    end if;
          end if;
     end loop;


     set_graph(root_graph, co);

     recycle(streams_properties);
     recycle(root_graph);
     edge_set_pkg.recycle(graphs_edges);
end update_root_edges;

-- For composite operators other than the root operator, this procedure
-- labels the source for input edges and the sink for output edges
-- as EXTERNAL
--
--        input streams:
--                  EXTERNAL -> input stream_name -> local sink operator
--
--        output streams:
--                  local source operator -> output stream_name -> EXTERNAL
--
procedure set_external_inputs_n_outputs(co: in out composite_operator;
                                   A, BASE, B, NEW_PSDL: psdl_program)
is
     parent_op_id: op_id;
     parent_op: composite_operator;
     new_graph, parent_graph: psdl_graph;
```

```
        input_streams, output_streams: type_declaration;
        graphs_edges: edge_set;
        streams_properties: init_map;
        streams_latency: millisec := undefined_time;
        external: op_id;

begin
        assign(new_graph, graph(co));
        assign(input_streams, inputs(co));
        assign(output_streams, outputs(co));
        assign(streams_properties, empty_init_map);
        set_op_id_operation_name(convert("EXTERNAL"), external);
        edge_set_pkg.assign(graphs_edges, edges(new_graph));

        -- for inputs, the sink will be local and the source will be EXTERNAL;
        for stream_name: psdl_id, tn: type_name in type_declaration_pkg.scan(input_streams)
        loop
                for e: edge in edge_set_pkg.scan(graphs_edges)
                loop
                        if eq(stream_name, e.stream_name) and not has_vertex(e.source, new_graph)
                        then
                                if not has_edge(external, e.sink, e.stream_name, new_graph) then
                                        merge_edge_attributes(streams_latency, streams_properties,
                                                        external,
                                                        e.sink, e.stream_name, name(co),
                                                        A, BASE, B);

                                        assign(new_graph, remove_edge(e, new_graph));
                                        assign(new_graph, add_edge(external, e.sink,
                                                        stream_name, new_graph,
                                                        streams_latency,
                                                        streams_properties));
                                else -- remove redundant externals for the stream_name with e.sink
                                        assign(new_graph, remove_edge(e, new_graph));
                                end if;
                        end if;
                end loop;

        end loop;
        recycle(streams_properties);
        streams_latency := undefined_time;

        -- for outputs, the sink will be EXTERNAL and the source will be local
        for stream_name: psdl_id, tn: type_name in type_declaration_pkg.scan(output_streams)
        loop
                for e: edge in edge_set_pkg.scan(graphs_edges)loop
                        if eq(stream_name, e.stream_name) and not has_vertex(e.sink, new_graph)
                        then
                                if not has_edge(e.source, external, e.stream_name, new_graph) then
                                        merge_edge_attributes(streams_latency, streams_properties,
                                                        e.source,
                                                        external, e.stream_name, name(co),
                                                        A, BASE, B);
```

```
                                    assign(new_graph, remove_edge(e, new_graph));
                                    assign(new_graph, add_edge(e.source, external,
                                                    stream_name, new_graph,
                                                    streams_latency,
                                                    streams_properties));
                            else -- remove redundant externals for the stream_name with e.source
                                    assign(new_graph, remove_edge(e, new_graph));
                            end if;
                    end if;
            end loop;
    end loop;

    set_graph(new_graph, co);

    recycle(streams_properties);
    recycle(new_graph);
    recycle(input_streams);
    recycle(output_streams);
    edge_set_pkg.recycle(graphs_edges);

end set_external_inputs_n_outputs;

-- copy operator's streams from one composite operator to another
procedure copy_streams(from_op: composite_operator;
                       to_op: in out composite_operator)
is
    to_graph: psdl_graph;
    data_streams: type_declaration;
        to_graph_edges: edge_set;
begin
    assign(to_graph, graph(to_op));
    assign(data_streams, streams(from_op));
        edge_set_pkg.assign( to_graph_edges, edges(to_graph));
    -- for an edge in the to_op graph that is also in the from_ops
    -- data streams set (str), copy it to to_ops data stream set
    for e: edge in edge_set_pkg.scan(to_graph_edges) loop
        for stream_name: psdl_id, tn: type_name in
                                    type_declaration_pkg.scan(data_streams)
            loop
            if eq(stream_name, e.stream_name) then
                if not member(stream_name, streams(to_op))
                                    and not member(stream_name, inputs(to_op))
                                    and not member(stream_name, outputs(to_op)) then
                    add_stream(stream_name, tn, to_op);
                end if;
            end if;
        end loop;
    end loop;
    recycle(to_graph);
    recycle(data_streams);
        edge_set_pkg.recycle(to_graph_edges);
end copy_streams;
```

```
-- recurse through composite operator graphs to finish reconstruction of composite
-- operators' specification and implementation graphs
procedure finish_composite_operator_construction(gr: psdl_graph;
                                    A, BASE, B, NEW_PSDL: psdl_program;
                                    co, new_root_co, merged_root_co: psdl_component)
is
        graphs_vertices: op_id_set;
        graphs_edges : edge_set;
        source_not_in_vertices, sink_not_in_vertices: Boolean := True;
        local_co: psdl_component;
        sum_of_children_smets: millisec := 0;
        copy_of_graph: psdl_graph;
        merged_type_name: type_name := null_type;
begin
        assign( graphs_vertices, vertices(gr));

        -- recurse down through composite operator graphs setting input and output
        -- stream attributes for composite operators. When this loop exits, any child
        -- composite operator for "operator_id" has been reconstructed and
        -- "operator_id's" graph has been updated and can be used to set "input" and
        -- "output" specification attributes

        for id: op_id in op_id_set_pkg.scan(graphs_vertices)
        loop
                local_co := get_definition(NEW_PSDL, id);
                if component_granularity(local_co) = composite then
                        assign(copy_of_graph, graph(local_co));
                        finish_composite_operator_construction(copy_of_graph, A, BASE, B,
                                        NEW_PSDL, local_co, new_root_co, merged_root_co);
                        recycle(copy_of_graph);
                end if;
        end loop;

-- if there is a source or sink for an edge and the source or sink is not in
-- the vertices set for the graph, then the edge is an input stream or output
-- stream ; so, assign the stream as an input stream or output stream for the
-- operator

        local_co := co;

        if not eq(local_co, new_root_co) then
                edge_set_pkg.assign( graphs_edges, edges(gr));

                for e: edge in edge_set_pkg.scan(graphs_edges) loop
                        source_not_in_vertices := True;
                        sink_not_in_vertices := True;

                        for id: op_id in op_id_set_pkg.scan(graphs_vertices) loop
                                if eq(e.source, id) then
                                        source_not_in_vertices := False;
                                end if;
                                if eq(e.sink, id) then
                                        sink_not_in_vertices := False;
                                end if;
```

```
                end loop;

                if source_not_in_vertices then
                        if not eq(convert("EXTERNAL"), base_name(e.source))
                        then
                                if not member(e.stream_name, inputs(local_co)) then

                                        merge_input_stream_type_names(
                                                        merged_type_name,
                                        name(local_co),
                                                        e.stream_name,
                                                        A, BASE, B);

                                        add_input(e.stream_name,
                                                merged_type_name, local_co);
                                end if;
                        end if;
                end if;

                if sink_not_in_vertices then
                        if not eq(convert("EXTERNAL"), base_name(e.sink))
                        then
                                if not member(e.stream_name, outputs(local_co)) then
                                        merge_output_stream_type_names(
                                                        merged_type_name,
                                        name(local_co),
                                                        e.stream_name,
                                                        A, BASE, B);

                                        add_output(e.stream_name,
                                                merged_type_name, local_co);
                                end if;
                        end if;
                end if;
        end loop;
        edge_set_pkg.recycle(graphs_edges);
end if;


-- copy over data streams from merged co corresponding to edges
-- in co's graph
copy_streams(merged_root_co, local_co);

if not eq(local_co, new_root_co) then
        update_parents_graph(local_co, A, BASE, B, NEW_PSDL);
        set_external_inputs_n_outputs(local_co, A, BASE, B, NEW_PSDL);
else
        update_root_edges(local_co, A, BASE, B, NEW_PSDL);
end if;

-- set_visible_timers(local_co);

-- merge axioms, implementation descriptions, informal descriptions,
```

```
        -- and states

        merge_composite_elements(A, BASE, B, local_co);

        recycle(graphs_vertices);


end finish_composite_operator_construction;


-- copy operator's timing constraints (period, fw, mcp, mrt) from one composite operator
-- to another
procedure copy_timing_constraints(operator_id: op_id; from_op: composite_operator;
                                   to_op: in out composite_operator)
is
begin
    set_period(operator_id, period(operator_id, from_op), to_op);

    set_finish_within(operator_id, finish_within(operator_id, from_op), to_op);

    set_minimum_calling_period(operator_id,
                                   minimum_calling_period(operator_id, from_op), to_op);

    set_maximum_response_time(operator_id,
                                   maximum_response_time(operator_id, from_op), to_op);

end copy_timing_constraints;



procedure copy_exception_triggers(operator_id: op_id;
    from_op: composite_operator; to_op: in out composite_operator)
is
    local_op_id: op_id := operator_id;
        excep_trigs: excep_trigger_map;
begin
        assign(excep_trigs, get_exception_trigger(from_op));

    for ex: excep_id, exprs: expression in excep_trigger_map_pkg.scan(excep_trigs) loop
        if eq(ex.op, local_op_id) then
            set_exception_trigger(ex,
                exception_trigger(local_op_id, ex.excep, from_op), to_op);
        end if;
    end loop;
        recycle(excep_trigs);

end copy_exception_triggers;

-- copy operator's control constraints (triggers, execution guards, output guards, and
-- exception triggers) from one composite operator to another
procedure copy_control_constraints(operator_id: op_id; gr: psdl_graph;
        from_op: composite_operator; to_op: in out composite_operator)
is
        guards: exec_guard_map;
```

```
    local_op_id: op_id := operator_id;
begin
    set_trigger(operator_id, get_trigger(operator_id, from_op), to_op);

    set_execution_guard(operator_id,
                        execution_guard(operator_id, from_op), to_op);

    for e: edge in edge_set_pkg.scan(edges(gr)) loop
        if eq(e.source, local_op_id) then
            set_output_guard(local_op_id, e.stream_name,
            output_guard(local_op_id, e.stream_name, from_op), to_op);
        end if;
    end loop;

        copy_exception_triggers(local_op_id, from_op, to_op);

end copy_control_constraints;


-- copy operator and corresponding edges from one psdl_graph operator to another
procedure copy_vertex_n_edges(op: op_id; from_graph: psdl_graph; to_graph: in out psdl_graph)
is
        local_op_id: op_id := op;
        from_graph_edges, to_graph_edges: edge_set;
begin
        -- copy vertex from from_graph to to_graph
        assign(to_graph, add_vertex(local_op_id, to_graph,
                        maximum_execution_time(local_op_id, from_graph),
                                get_properties(local_op_id, from_graph)));

        edge_set_pkg.assign(from_graph_edges, edges(from_graph));
        edge_set_pkg.assign(to_graph_edges, edges(to_graph));

-- copy the edge from from_graph to to_graph if op is either source or sink for
-- edge in from_graph
        for e: edge in edge_set_pkg.scan(from_graph_edges) loop
                if eq(e.source, local_op_id) or eq(e.sink, local_op_id) then
                        if not member(e, to_graph_edges) then
                                assign(to_graph,
                                        add_edge(e.source, e.sink, e.stream_name, to_graph,
                                        latency(e.source, e.sink, e.stream_name, from_graph),
                                        get_properties(e.source, e.sink, e.stream_name,
                                                        from_graph)));
                        end if;
                end if;
        end loop;

        edge_set_pkg.recycle(from_graph_edges);
        edge_set_pkg.recycle(to_graph_edges);
end copy_vertex_n_edges;


procedure copy_timer_operations(op: op_id; to_node: in out composite_operator;
```

```
                              from_node: composite_operator)
is
    timer_ops: timer_op_set;
begin
        assign(timer_ops, timer_operations(op, from_node));
    set_timer_op(op, timer_ops, to_node);

end copy_timer_operations;

end reconstruct_prototype_utilities_pkg;
```

4. ancestor_chains_pkg

```
with generic_map_pkg;
with psdl_id_pkg; use psdl_id_pkg;
with extended_ancestor_pkg; use extended_ancestor_pkg;
package ancestor_chains_pkg is

package ancestor_chains_map_inst_pkg is
        new generic_map_pkg(key => psdl_id, result => extended_ancestor,
                eq_key => eq, eq_res => eq,
                average_size => 100);

subtype ancestor_chains is ancestor_chains_map_inst_pkg.map;

-- Returns an empty ancestor_chains.
function empty_ancestor_chains return ancestor_chains;

procedure put_ancestor_chains(ea_map: ancestor_chains);

end ancestor_chains_pkg;


with text_io; use text_io;
with extended_ancestor_pkg; use extended_ancestor_pkg;
package body ancestor_chains_pkg is

-- Returns an empty ancestor_chains.
function empty_ancestor_chains return ancestor_chains is
        ac : ancestor_chains;
begin
        ancestor_chains_map_inst_pkg.create(null_ancestor, ac);
        return ac;
end empty_ancestor_chains;

procedure  put_ancestor_chains(ea_map: ancestor_chains)
is
begin
    for N: psdl_id, ea: extended_ancestor in
                ancestor_chains_map_inst_pkg.scan(ea_map)
    loop
            put(convert(N)); put("'s ancestor chain: ");
            put_ancestor(ea);
    end loop;
end put_ancestor_chains;

end ancestor_chains_pkg;
```

APPENDIX B. EXTENSIONS AND CHANGES TO PSDL_TYPE

This appendix lists the changes and extensions to the PSDL_TYPE Abstract Data Type made during design and implementation of the Decomposition Recovery Extension.

Changed Source Files:

PSDL_TYPE/psdl_ct_s.a
PSDL_TYPE/psdl_graph_b.g
PSDL_TYPE/psdl_graph_s.a
PSDL_TYPE/psdl_type_b.g
PSDL_TYPE/psdl_type_s.a
PSDL_TYPE/INSTANTIATIONS/psdl_id_seq.a

PSDL_TYPE/psdl_ct_s.a

The following was added:

```
function length(s: psdl_id_sequence) return natural
  renames psdl_id_seq_pkg.length;
procedure recycle(s: in out psdl_id_sequence)
  renames psdl_id_seq_pkg.recycle;
procedure fetch(s1: psdl_id_sequence; low, high: natural; s: in out psdl_id_sequence)
  renames psdl_id_seq_pkg.fetch;
```

PSDL_TYPE/psdl_graph_s.a

The following was added:

```
-- remove_edge: removes a directed edge from source to sink in g.
function remove_edge(e: edge;
           g: psdl_graph; latency: millisec := undefined_time;
           properties: init_map := empty_init_map)
       return psdl_graph;

function has_edge(source, sink: op_id; stream_name: psdl_id; g: psdl_graph)
  return boolean;
  -- Returns true if and only if there exists an edge
  -- from vertex source to vertex sink in g with stream_name.
```

PSDL_TYPE/psdl_graph_b.g

The following was added:

```
-- remove_edge: removes a directed edge from source to sink in g.
 function remove_edge(e: edge;
              g: psdl_graph; latency: millisec := undefined_time;
              properties: init_map := empty_init_map)
        return psdl_graph is
  h: psdl_graph;
 begin
  assign(h, g);
  edge_set_pkg.remove(e, h.edges);
  latency_map_pkg.remove(e, h.latency);
  remove(e, h.edge_properties);
  return h;
 end remove_edge;

 -- Returns true if and onlocal_sink if there exists
 -- an edge from vertex source to vertex sink in g.
 function has_edge(source, sink: op_id; stream_name: psdl_id; g: psdl_graph)
 return boolean
 is
  local_source: op_id := source; -- Local copy to avoid compiler bug.
  local_sink: op_id := sink; -- Local copy to avoid compiler bug.
  local_stream_name: psdl_id := stream_name; -- Local copy to avoid compiler bug.
  result: boolean := false;
 begin
  for e: edge in edge_set_pkg.scan(g.edges) loop
   if eq(e.source, local_source) and eq(e.stream_name, local_stream_name) and
                                          eq(e.sink, local_sink)
   then result := true; exit; end if;
  end loop;
  return(result);
 end has_edge;
```

PSDL_TYPE/psdl_type_s.a

The following was added:

```
-- binds a timer_op_set to a composite operator
procedure set_timer_op(o: op_id; timer_ops: timer_op_set;
                       co: in out composite_operator);


function get_exception_trigger(op: composite_operator)return excep_trigger_map;

procedure set_exception_trigger(e: excep_id; ex: expression;
                                op: composite_operator);


procedure set_informal_description(inf_desc: text; co: psdl_component);

procedure set_axioms(ax: text;  co: psdl_component);

procedure set_implementation_description(impl_desc: text; co: psdl_component);
```

PSDL_TYPE/psdl_type_b.g

The following was changed:

In set_graph(), the statement "co.g := g;" was causing crashes. It was replaced with the statement "assign(co.g, g);" which appears to have fixed the problem.

```
-- co.g := g;
assign(co.g, g);
```

The following was added:

```
-- adds a timer_op_set to a composite operator
  procedure set_timer_op(o: op_id; timer_ops: timer_op_set;
                         co: in out composite_operator) is
  begin
    if co = null_component then raise undefined_component; end if;

    bind(o, timer_ops, co.tim_op);
  end set_timer_op;
```

```
function get_exception_trigger(op: composite_operator) return excep_trigger_map
is
begin
        return op.et;
end get_exception_trigger;

procedure set_exception_trigger(e: excep_id; ex: expression;
                                                  op: composite_operator)
is
begin
      if not member(e, op.et) then
            bind(e, ex, op.et);
      end if;
end set_exception_trigger;


procedure set_informal_description(inf_desc: text; co: psdl_component)
is
begin
      if co = null_component then raise undefined_component; end if;

      co.inf_desc := inf_desc;

end set_informal_description;

procedure set_axioms(ax: text;  co: psdl_component)
is
begin
      if co = null_component then raise undefined_component; end if;

      co.ax := ax;
end set_axioms;

procedure set_implementation_description(impl_desc: text; co: psdl_component)
is
begin
   if co = null_component then raise undefined_component; end if;
   co.impl_desc := impl_desc;

end set_implementation_description;
```

PSDL_TYPE/INSTANTIATIONS/psdl_id_seq.a

The following was added:

```
function length(s: psdl_id_sequence) return natural
  renames psdl_id_seq_inst_pkg.length;
procedure recycle(s: in out psdl_id_sequence)
  renames psdl_id_seq_inst_pkg.recycle;
procedure fetch(s1: psdl_id_sequence; low, high: natural; s: in out psdl_id_sequence)
  renames psdl_id_seq_inst_pkg.fetch;
```

APPENDIX C. TEST-CASES

This appendix describes test-cases used to test *ancestor chain* merge and PSDL prototype decomposition structure reconstruction. For each test-case, a brief description is given followed by a listing of test-driver source code and test output.

For most of these test-cases, PSDL prototype specification files (Expanded-Merged prototype, Change A, BASE, Change B) were used a input. In order to keep the size of this appendix manageable, these file are not included here. However, they are described, and they are available from the author upon request (keesling@nosc.mil).

Test-Case: test_merge_chains

Used to demonstrate conflict-free *ancestor chain* merges, as well as conflicting *ancestor chain* merges with accompanying conflict reporting and resolution. The actual test-cases are hard-coded into the test driver.

Test-Driver: test_merge_chains

```
with text_io; use text_io;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with extended_ancestor_pkg; use extended_ancestor_pkg;
with decompose_graph_pkg; use decompose_graph_pkg;
with ancestor_chains_pkg; use ancestor_chains_pkg;
procedure test_merge_chains
is

        ea_1, ea_2, ea_3: extended_ancestor;

        procedure merge_test(A, BASE, B: extended_ancestor; atomic_op: psdl_id)
        is
                resolve_chain, merge: extended_ancestor;
        begin
                put("A:    ");
                put_ancestor(A);
                put("BASE: ");
                put_ancestor(BASE);
                put("B:    ");
                put_ancestor(B);
                merge_ancestor_chains(A, BASE, B, merge);
                if type_of_ancestor(merge) = improper then
                        put_conflict_message(atomic_op, merge);
```

```
                    resolve_chain := resolve_conflict(merge);
                    put_line("CONFLICT RESOLVED");
                    recycle_extended_ancestor(merge);
                    merge := resolve_chain;
            end if;
            put("MERGE = ");
            put_ancestor(merge);
            put_line(" ");
            recycle_extended_ancestor(merge);
        end merge_test;
begin

        ea_1 := build_proper_ancestor(empty);
        append_ancestor(ea_1, convert("root_op"));
        append_ancestor(ea_1, convert("op_1"));
        append_ancestor(ea_1, convert("op_2"));
        append_ancestor(ea_1, convert("op_3"));
        append_ancestor(ea_1, convert("op_4"));

        ea_2 := build_proper_ancestor(empty);
        append_ancestor(ea_2, convert("root_op"));
        append_ancestor(ea_2, convert("op_1"));
        append_ancestor(ea_2, convert("op_2"));
        append_ancestor(ea_2, convert("op_3"));
        append_ancestor(ea_2, convert("op_4"));
        append_ancestor(ea_2, convert("op_5"));
        append_ancestor(ea_2, convert("op_6"));

        ea_3 := build_proper_ancestor(empty);
        append_ancestor(ea_3, convert("root_op"));
        append_ancestor(ea_3, convert("op_1"));
        append_ancestor(ea_3, convert("op_2"));
        append_ancestor(ea_3, convert("op_3"));
        append_ancestor(ea_3, convert("op_4"));
        append_ancestor(ea_3, convert("op_5"));
        append_ancestor(ea_3, convert("op_6"));
        append_ancestor(ea_3, convert("op_7"));

        put_line("A = BASE /= B");
        merge_test(ea_1, ea_1, ea_2, convert("atomic_op"));

        put_line("A /= B = BASE");
        merge_test(ea_1, ea_2, ea_2, convert("atomic_op"));
```

```
put_line("A = B /= BASE");
merge_test(ea_1, ea_2, ea_1, convert("atomic_op"));

put_line("A /= BASE /= B");
merge_test(ea_2, ea_1, ea_3, convert("atomic_op"));

recycle_extended_ancestor(ea_1);

ea_1 := build_proper_ancestor(empty);
append_ancestor(ea_1, convert("root_op"));
append_ancestor(ea_1, convert("op_1"));
append_ancestor(ea_1, convert("op_2"));
append_ancestor(ea_1, convert("op_3"));
append_ancestor(ea_1, convert("op_8"));
append_ancestor(ea_1, convert("op_4"));

put_line("A /= BASE /= B");
merge_test(ea_2, ea_1, ea_3, convert("atomic_op"));

put_line("A /= BASE /= B");
merge_test(ea_1, ea_2, ea_3, convert("atomic_op"));

put_line("A /= BASE /= B");
merge_test(ea_3, ea_1, ea_2, convert("atomic_op"));

put_line("A /= BASE /= B");
merge_test(ea_3, ea_2, ea_1, convert("atomic_op"));

recycle_extended_ancestor(ea_1);
recycle_extended_ancestor(ea_2);
recycle_extended_ancestor(ea_3);


ea_1 := build_proper_ancestor(empty);
append_ancestor(ea_1, convert("root_op"));
append_ancestor(ea_1, convert("op_1"));
append_ancestor(ea_1, convert("op_2"));
append_ancestor(ea_1, convert("op_3"));
append_ancestor(ea_1, convert("op_4"));
append_ancestor(ea_1, convert("op_51"));
append_ancestor(ea_1, convert("op_6"));
append_ancestor(ea_1, convert("op_7"));

ea_2 := build_proper_ancestor(empty);
append_ancestor(ea_2, convert("root_op"));
```

```
append_ancestor(ea_2, convert("op_1"));
append_ancestor(ea_2, convert("op_2"));
append_ancestor(ea_2, convert("op_3"));
append_ancestor(ea_2, convert("op_4"));
append_ancestor(ea_2, convert("op_5"));
append_ancestor(ea_2, convert("op_6"));
append_ancestor(ea_2, convert("op_7"));

ea_3 := build_proper_ancestor(empty);
append_ancestor(ea_3, convert("root_op"));
append_ancestor(ea_3, convert("op_1"));
append_ancestor(ea_3, convert("op_2"));
append_ancestor(ea_3, convert("op_3"));
append_ancestor(ea_3, convert("op_4"));
append_ancestor(ea_3, convert("op_5"));
append_ancestor(ea_3, convert("op_63"));
append_ancestor(ea_3, convert("op_7"));


put_line("A /= BASE /= B");
merge_test(ea_1, ea_2, ea_3, convert("atomic_op"));

recycle_extended_ancestor(ea_1);
recycle_extended_ancestor(ea_2);
recycle_extended_ancestor(ea_3);

ea_1 := build_proper_ancestor(empty);
append_ancestor(ea_1, convert("root_op"));

ea_2 := build_proper_ancestor(empty);
append_ancestor(ea_2, convert("root_op"));
append_ancestor(ea_2, convert("op_1"));
append_ancestor(ea_2, convert("op_2"));
append_ancestor(ea_2, convert("op_3"));
append_ancestor(ea_2, convert("op_4"));
append_ancestor(ea_2, convert("op_5"));
append_ancestor(ea_2, convert("op_6"));
append_ancestor(ea_2, convert("op_7"));

ea_3 := build_proper_ancestor(empty);
append_ancestor(ea_3, convert("root_op"));
append_ancestor(ea_3, convert("op_1"));
append_ancestor(ea_3, convert("op_2"));
append_ancestor(ea_3, convert("op_3"));
append_ancestor(ea_3, convert("op_4"));
```

```
append_ancestor(ea_3, convert("op_5"));


put_line("A = BASE /= B");
merge_test(ea_1, ea_1, ea_3, convert("atomic_op"));

put_line("A = B = root_op /= BASE");
merge_test(ea_1, ea_3, ea_1, convert("atomic_op"));

put_line("A = B = BASE");
merge_test(ea_3, ea_3, ea_3, convert("atomic_op"));

put_line("A /= B /= BASE");
merge_test(ea_1, ea_3, ea_2, convert("atomic_op"));

put_line("A /= B /= BASE");
merge_test(ea_1, ea_2, ea_3, convert("atomic_op"));

put_line("A /= BASE = EMPTY /= B");
merge_test(ea_2, empty_extended_ancestor, ea_3, convert("atomic_op"));

put_line("A = BASE = EMPTY /= B");
merge_test(empty_extended_ancestor, empty_extended_ancestor, ea_3,
                                             convert("atomic_op"));

put_line("A = EMPTY /= BASE /= B");
merge_test(empty_extended_ancestor, ea_2, ea_3, convert("atomic_op"));

put_line("A = EMPTY /= BASE /= B");
merge_test(empty_extended_ancestor, ea_3, ea_2, convert("atomic_op"));

recycle_extended_ancestor(ea_3);
ea_3 := build_proper_ancestor(empty);
append_ancestor(ea_3, convert("root_op"));
append_ancestor(ea_3, convert("op_1"));
append_ancestor(ea_3, convert("op_2"));
append_ancestor(ea_3, convert("op_3"));
append_ancestor(ea_3, convert("op_4"));
append_ancestor(ea_3, convert("op_9"));

put_line("A = EMPTY /= BASE /= B");
merge_test(empty_extended_ancestor, ea_3, ea_2, convert("atomic_op"));

put_line("A = EMPTY /= BASE /= B");
merge_test(empty_extended_ancestor, ea_2, ea_3, convert("atomic_op"));
```

```
        recycle_extended_ancestor(ea_1);
        recycle_extended_ancestor(ea_2);
        recycle_extended_ancestor(ea_3);
end test_merge_chains;
```

Test Output: test_merge_chains

```
A = BASE /= B
A:   root_op->op_1->op_2->op_3->op_4
BASE: root_op->op_1->op_2->op_3->op_4
B:   root_op->op_1->op_2->op_3->op_4->op_5->op_6
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6

A /= B = BASE
A:   root_op->op_1->op_2->op_3->op_4
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6
B:   root_op->op_1->op_2->op_3->op_4->op_5->op_6
MERGE = root_op->op_1->op_2->op_3->op_4

A = B /= BASE
A:   root_op->op_1->op_2->op_3->op_4
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6
B:   root_op->op_1->op_2->op_3->op_4
MERGE = root_op->op_1->op_2->op_3->op_4

A /= BASE /= B
A:   root_op->op_1->op_2->op_3->op_4->op_5->op_6
BASE: root_op->op_1->op_2->op_3->op_4
B:   root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7

A /= BASE /= B
A:   root_op->op_1->op_2->op_3->op_4->op_5->op_6
BASE: root_op->op_1->op_2->op_3->op_8->op_4
B:   root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7

A /= BASE /= B
A:   root_op->op_1->op_2->op_3->op_8->op_4
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6
B:   root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
ONE OR MORE CONFLICTS IN ANCESTOR CHAIN RECOVERY FOR: atomic_op
<root_op->op_1->op_2->op_3->op_8->op_4>
[<root_op->op_1->op_2->op_3->op_4->op_5->op_6>]
```

<root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7> =
<root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7> U
<root_op->op_1->op_2->op_3->op_8->op_4>=
(***conflict***) =
<root_op->op_1->op_2->op_3(op_4->op_5->op_6->op_7 U op_8->op_4)>

CONFLICT RESOLVED
MERGE = root_op->op_1->op_2->op_3

A /= BASE /= B
A: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
BASE: root_op->op_1->op_2->op_3->op_8->op_4
B: root_op->op_1->op_2->op_3->op_4->op_5->op_6
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7

A /= BASE /= B
A: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6
B: root_op->op_1->op_2->op_3->op_8->op_4
ONE OR MORE CONFLICTS IN ANCESTOR CHAIN RECOVERY FOR: atomic_op
<root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7>
[<root_op->op_1->op_2->op_3->op_4->op_5->op_6>]
<root_op->op_1->op_2->op_3->op_8->op_4> =
<root_op->op_1->op_2->op_3->op_8->op_4> U
<root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7>=
(***conflict***) =
<root_op->op_1->op_2->op_3(op_8->op_4 U op_4->op_5->op_6->op_7)>

CONFLICT RESOLVED
MERGE = root_op->op_1->op_2->op_3

A /= BASE /= B
A: root_op->op_1->op_2->op_3->op_4->op_51->op_6->op_7
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
B: root_op->op_1->op_2->op_3->op_4->op_5->op_63->op_7
ONE OR MORE CONFLICTS IN ANCESTOR CHAIN RECOVERY FOR: atomic_op
<root_op->op_1->op_2->op_3->op_4->op_51->op_6->op_7>
[<root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7>]
<root_op->op_1->op_2->op_3->op_4->op_5->op_63->op_7> =
<root_op->op_1->op_2->op_3->op_4->op_5->op_63->op_7> U
<root_op->op_1->op_2->op_3->op_4->op_51->op_6->op_7>=
(***conflict***) =
<root_op->op_1->op_2->op_3->op_4(op_5->op_63->op_7 U op_51->op_6->op_7)>

CONFLICT RESOLVED

MERGE = root_op->op_1->op_2->op_3->op_4

A = BASE /= B
A: root_op
BASE: root_op
B: root_op->op_1->op_2->op_3->op_4->op_5
MERGE = root_op->op_1->op_2->op_3->op_4->op_5

A = B = root_op /= BASE
A: root_op
BASE: root_op->op_1->op_2->op_3->op_4->op_5
B: root_op
MERGE = root_op

A = B = BASE
A: root_op->op_1->op_2->op_3->op_4->op_5
BASE: root_op->op_1->op_2->op_3->op_4->op_5
B: root_op->op_1->op_2->op_3->op_4->op_5
MERGE = root_op->op_1->op_2->op_3->op_4->op_5

A /= B /= BASE
A: root_op
BASE: root_op->op_1->op_2->op_3->op_4->op_5
B: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7

A /= B /= BASE
A: root_op
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
B: root_op->op_1->op_2->op_3->op_4->op_5
MERGE = root_op

A /= BASE = EMPTY /= B
A: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
BASE: EMPTY CHAIN
B: root_op->op_1->op_2->op_3->op_4->op_5
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7

A = BASE = EMPTY /= B
A: EMPTY CHAIN
BASE: EMPTY CHAIN
B: root_op->op_1->op_2->op_3->op_4->op_5
MERGE = root_op->op_1->op_2->op_3->op_4->op_5

A = EMPTY /= BASE /= B

A: EMPTY CHAIN
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
B: root_op->op_1->op_2->op_3->op_4->op_5
MERGE = EMPTY CHAIN

A = EMPTY /= BASE /= B
A: EMPTY CHAIN
BASE: root_op->op_1->op_2->op_3->op_4->op_5
B: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7

A = EMPTY /= BASE /= B
A: EMPTY CHAIN
BASE: root_op->op_1->op_2->op_3->op_4->op_9
B: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
MERGE = root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7

A = EMPTY /= BASE /= B
A: EMPTY CHAIN
BASE: root_op->op_1->op_2->op_3->op_4->op_5->op_6->op_7
B: root_op->op_1->op_2->op_3->op_4->op_9
MERGE = root_op->op_1->op_2->op_3->op_4->op_9

Test-Case: test_merge_demo

This uses the test cases that Dr. Dampier apparently used to demo his merge tool... used again here to demonstrate that prototypes with no decomposition structure (save the single root composite) could pass through decompose_graph resulting in a correctly formed prototype. It also demonstrates that text descriptions are recovered for composites.

Test-Driver: test_merge_demo

```
with TEXT_IO; use TEXT_IO;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_io; use psdl_io;
with extended_ancestor_pkg; use extended_ancestor_pkg;
with ancestor_chains_pkg; use ancestor_chains_pkg;
with decompose_graph_pkg; use decompose_graph_pkg;
procedure test_merge_demo is
        TESTFILE: FILE_TYPE;
```

```
  NEW_PSDL, A_PSDL, BASE_PSDL, B_PSDL, MERGE: psdl_program;
root_op: psdl_id;

  ancestors: ancestor_chains;

MERGE_CHAIN: extended_ancestor := null_ancestor;

begin
    OPEN(TESTFILE,IN_FILE,"merge.demo.MERGE.psdl");
    assign(MERGE,empty_psdl_program);

    put_line("getting change MERGE prototype file!");
    get(TESTFILE,MERGE);

    CLOSE(TESTFILE);
    -- put(MERGE);

    OPEN(TESTFILE,IN_FILE,"merge.demo.A.psdl");
    assign(A_PSDL,empty_psdl_program);

    put_line("getting change A prototype file!");
    get(TESTFILE,A_PSDL);

    CLOSE(TESTFILE);
    -- put(A_PSDL);

    OPEN(TESTFILE,IN_FILE,"merge.demo.Base.psdl");
    assign(BASE_PSDL,empty_psdl_program);

    put_line("getting change BASE prototype file!");
    get(TESTFILE,BASE_PSDL);

    CLOSE(TESTFILE);
    -- put(BASE_PSDL);

    OPEN(TESTFILE,IN_FILE,"merge.demo.B.psdl");
    assign(B_PSDL,empty_psdl_program);

    put_line("getting change B prototype file!");
    get(TESTFILE,B_PSDL);

    CLOSE(TESTFILE);
    -- put(B_PSDL);

    decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE, NEW_PSDL);
```

```
        -- need the root operator for find_ancestor_chain.
        root_op := find_root(NEW_PSDL);
        put_line(convert(root_op));

        put(NEW_PSDL);
        ancestor_chains_map_inst_pkg.assign(ancestors, empty_ancestor_chains);
      for id: psdl_id, c : psdl_component in psdl_program_map_pkg.scan(NEW_PSDL)
        loop
              if component_category(c) = psdl_operator then
                    if component_granularity(c) = atomic then
                    MERGE_CHAIN := find_ancestor_chain(id, root_op,
                                                                     NEW_PSDL);
                          ancestor_chains_map_inst_pkg.bind(id, MERGE_CHAIN,
                                                                     ancestors);
                    end if;
              end if;
      end loop;

        put_ancestor_chains(ancestors);
        ancestor_chains_map_inst_pkg.recycle(ancestors);
end test_merge_demo;
```

Test-Output: test_merge_demo

```
getting change MERGE prototype file!
getting change A prototype file!
getting change BASE prototype file!
getting change B prototype file!
D HAS EMPTY MERGED CHAIN, POSSIBLE MERGE CONFLICT
ASSIGNING ROOT OPERATOR AS PARENT
A's ancestor chain: DEMO2
B's ancestor chain: DEMO2
E's ancestor chain: DEMO2
D's ancestor chain: DEMO2
DEMO2
OPERATOR DEMO2
 SPECIFICATION
  DESCRIPTION { This is the psdl program used in the 2nd demo of the change
     merge tool.  }
 END

 IMPLEMENTATION
  GRAPH
   VERTEX A
```

```
    VERTEX B
    VERTEX E
    VERTEX D

    EDGE AOUT A -> B
    EDGE DIN A -> D
    EDGE BOUT B -> E

   CONTROL CONSTRAINTS
    OPERATOR A
    OPERATOR B
    OPERATOR E
    OPERATOR D
  DESCRIPTION {  This implementation is not real.  It does absolutely nothing. }
 END

OPERATOR A
 SPECIFICATION
  OUTPUT
   AOUT: t1,
   CIN: t2
  DESCRIPTION { nada }
 END

 IMPLEMENTATION ADA A
 END

OPERATOR B
 SPECIFICATION
  INPUT
   AOUT: t1
  OUTPUT
   BOUT: t3
  DESCRIPTION { nada }
 END

 IMPLEMENTATION ADA B
 END

OPERATOR E
 SPECIFICATION
  INPUT
   BOUT: t3
  DESCRIPTION { nada }
 END
```

```
 IMPLEMENTATION ADA E
 END

OPERATOR D
 SPECIFICATION
  INPUT
   DIN: t2
  DESCRIPTION { nada }
 END

 IMPLEMENTATION ADA D
 END
```

A's ancestor chain: DEMO2
B's ancestor chain: DEMO2
E's ancestor chain: DEMO2
D's ancestor chain: DEMO2

Test-Case: test_dg_1

This test-case demonstrates correctness of merge and prototype decomposition structure recovery for non-overlapping, or disjoint, hierarchical changes. For this test-case, I edited existing prototype file atacms.psdl to create changes A & B, and the BASE version. I ran atacms.psdl through the expander to create a flattened version to use as MERGE: atacms_ex.psdl. I removed composite operator gui_in (and all associated streams and vertices) from atacms.psdl to produce atacms.A.psdl. I then removed composite operator gui_out (and all associated streams and vertices) from atacms.psdl to produce atacms.B.psdl. I then removed both gui_out and gui_in (and all associated streams and vertices) from atacms.psdl to produce atacms.BASE.psdl.

Test-Driver: test_dg_1

```
with TEXT_IO; use TEXT_IO;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_io; use psdl_io;
with extended_ancestor_pkg; use extended_ancestor_pkg;
with ancestor_chains_pkg; use ancestor_chains_pkg;
with decompose_graph_pkg; use decompose_graph_pkg;
procedure test_dg_1 is
        TESTFILE: FILE_TYPE;
        NEW_PSDL, A_PSDL, BASE_PSDL, B_PSDL, MERGE: psdl_program;
      root_op: psdl_id;
```

```
    ancestors: ancestor_chains;

  MERGE_CHAIN: extended_ancestor := null_ancestor;

begin
    OPEN(TESTFILE,IN_FILE,"atacms_ex.psdl");
    assign(MERGE,empty_psdl_program);

    put_line("getting MERGE prototype file!");
    get(TESTFILE,MERGE);

    CLOSE(TESTFILE);
    -- put(MERGE);

    OPEN(TESTFILE,IN_FILE,"atacms.A.psdl");
    assign(A_PSDL,empty_psdl_program);

    put_line("getting change A prototype file!");
    get(TESTFILE,A_PSDL);

    CLOSE(TESTFILE);
    -- put(A_PSDL);

    OPEN(TESTFILE,IN_FILE,"atacms.Base.psdl");
    assign(BASE_PSDL,empty_psdl_program);

    put_line("getting change BASE prototype file!");
    get(TESTFILE,BASE_PSDL);

    CLOSE(TESTFILE);
    -- put(BASE_PSDL);

    OPEN(TESTFILE,IN_FILE,"atacms.B.psdl");
    assign(B_PSDL,empty_psdl_program);

    put_line("getting change B prototype file!");
    get(TESTFILE,B_PSDL);

    CLOSE(TESTFILE);
    -- put(B_PSDL);

    decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE, NEW_PSDL);

    -- need the root operator for find_ancestor_chain.
```

```
        root_op := find_root(NEW_PSDL);
        put_line(convert(root_op));

        put(NEW_PSDL);
        ancestor_chains_map_inst_pkg.assign(ancestors, empty_ancestor_chains);
      for id: psdl_id, c : psdl_component in psdl_program_map_pkg.scan(NEW_PSDL)
        loop
              if component_category(c) = psdl_operator then
                    if component_granularity(c) = atomic then
                    MERGE_CHAIN := find_ancestor_chain(id, root_op,
                                                       NEW_PSDL);
                          ancestor_chains_map_inst_pkg.bind(id, MERGE_CHAIN,
                                                             ancestors);
                    end if;
              end if;
      end loop;

        put_ancestor_chains(ancestors);
        ancestor_chains_map_inst_pkg.recycle(ancestors);
end test_dg_1;
```

Test-Output: test_dg_1

```
getting MERGE prototype file!
getting change A prototype file!
getting change BASE prototype file!
getting change B prototype file!
asas_op's ancestor chain: atacms->command_station_op
choose_inputs's ancestor chain: atacms->gui_in
cmds_out's ancestor chain: atacms->gui_out
cnr_link_op's ancestor chain: atacms
ctoc_op's ancestor chain: atacms->command_station_op
grnd_stat_mod_op's ancestor chain: atacms->command_station_op
gui_input_event_monitor's ancestor chain: atacms->gui_in
jstars_op's ancestor chain: atacms
lan1_link_op's ancestor chain: atacms->command_station_op
lan2_link_op's ancestor chain: atacms->command_station_op
scdl_link_op's ancestor chain: atacms
shooter_op's ancestor chain: atacms
target_emitter_op's ancestor chain: atacms
atacms
OPERATOR atacms
  SPECIFICATION
    STATES gui_in_str: my_unit INITIALLY pause
  END
```

```
IMPLEMENTATION
 GRAPH
  VERTEX command_station_op
  VERTEX gui_in
  VERTEX gui_out
  VERTEX cnr_link_op: 50 MS
  VERTEX jstars_op: 500 MS
  VERTEX scdl_link_op: 50 MS
  VERTEX shooter_op: 50 MS
  VERTEX target_emitter_op: 500 MS

  EDGE fire_cmd4_str cnr_link_op -> shooter_op
  EDGE emission_str target_emitter_op -> jstars_op
  EDGE target_array1_str jstars_op -> scdl_link_op
  EDGE gui_in_str gui_in -> command_station_op
  EDGE fire_cmd3_str command_station_op -> cnr_link_op
  EDGE gui_in_str gui_in -> jstars_op
  EDGE target_array2_str scdl_link_op -> command_station_op
  EDGE gui_out_str shooter_op -> gui_out
  EDGE gui_in_str gui_in -> target_emitter_op

 DATA STREAM
  fire_cmd4_str: target_data,
  fire_cmd3_str: target_data,
  emission_str: target_emitter_array,
  target_array1_str: jstars_array,
  target_array2_str: jstars_array,
  gui_out_str: target_data
 CONTROL CONSTRAINTS
  OPERATOR command_station_op
  OPERATOR gui_in
  OPERATOR gui_out
  OPERATOR cnr_link_op
   TRIGGERED BY SOME fire_cmd3_str
  OPERATOR jstars_op
   TRIGGERED IF (gui_in_str /= my_unit.pause)
   PERIOD 8000 MS
  OPERATOR scdl_link_op
   TRIGGERED BY SOME target_array1_str
  OPERATOR shooter_op
   TRIGGERED BY SOME fire_cmd4_str
  OPERATOR target_emitter_op
   TRIGGERED IF (gui_in_str /= my_unit.pause)
   PERIOD 16000 MS
```

```
 END

OPERATOR command_station_op
 SPECIFICATION
  INPUT
   gui_in_str: my_unit,
   target_array2_str: jstars_array
  OUTPUT
   fire_cmd3_str: target_data
 END

 IMPLEMENTATION
  GRAPH
   VERTEX asas_op: 200 MS
   VERTEX ctoc_op: 50 MS
   VERTEX grnd_stat_mod_op: 50 MS
   VERTEX lan1_link_op: 50 MS
   VERTEX lan2_link_op: 50 MS

   EDGE fire_cmd1_str asas_op -> lan2_link_op
   EDGE target_array4_str lan1_link_op -> asas_op
   EDGE fire_cmd2_str lan2_link_op -> ctoc_op
   EDGE target_array3_str grnd_stat_mod_op -> lan1_link_op
   EDGE gui_in_str EXTERNAL -> asas_op
   EDGE target_array2_str : 5000 MS EXTERNAL -> grnd_stat_mod_op
   EDGE fire_cmd3_str ctoc_op -> EXTERNAL

  DATA STREAM
   fire_cmd1_str: target_data,
   target_array4_str: grnd_stat_mod_array,
   fire_cmd2_str: target_data,
   target_array3_str: grnd_stat_mod_array
  CONTROL CONSTRAINTS
   OPERATOR asas_op
    TRIGGERED IF (gui_in_str /= my_unit.pause)
    PERIOD 4000 MS
   OPERATOR ctoc_op
    TRIGGERED BY SOME fire_cmd2_str
   OPERATOR grnd_stat_mod_op
    TRIGGERED BY SOME target_array2_str
   OPERATOR lan1_link_op
    TRIGGERED BY SOME target_array3_str
   OPERATOR lan2_link_op
    TRIGGERED BY SOME fire_cmd1_str
 END
```

```
OPERATOR asas_op
 SPECIFICATION
  INPUT
   gui_in_str: my_unit,
   target_array4_str: grnd_stat_mod_array
  OUTPUT
   fire_cmd1_str: target_data
  MAXIMUM EXECUTION TIME 200 MS
 END

 IMPLEMENTATION ADA asas_op
 END

OPERATOR gui_in
 SPECIFICATION
  OUTPUT
   gui_in_str: my_unit
 END

 IMPLEMENTATION
  GRAPH
   VERTEX choose_inputs: 200 MS
   VERTEX gui_input_event_monitor: 200 MS

   EDGE gui_in_str choose_inputs -> EXTERNAL

  CONTROL CONSTRAINTS
   OPERATOR choose_inputs
    PERIOD 2000 MS
   OPERATOR gui_input_event_monitor
 END

OPERATOR choose_inputs
 SPECIFICATION
  OUTPUT
   gui_in_str: my_unit
  MAXIMUM EXECUTION TIME 200 MS
 END

 IMPLEMENTATION ADA choose_inputs
 END

OPERATOR gui_out
 SPECIFICATION
```

```
  INPUT
    gui_out_str: target_data
 END

 IMPLEMENTATION
  GRAPH
   VERTEX cmds_out

   EDGE gui_out_str EXTERNAL -> cmds_out

  CONTROL CONSTRAINTS
   OPERATOR cmds_out
    TRIGGERED BY SOME gui_out_str
 END

OPERATOR cmds_out
 SPECIFICATION
  INPUT
   gui_out_str: target_data
 END

 IMPLEMENTATION ADA cmds_out
 END

OPERATOR cnr_link_op
 SPECIFICATION
  INPUT
   fire_cmd3_str: target_data
  OUTPUT
   fire_cmd4_str: target_data
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA cnr_link_op
 END

OPERATOR ctoc_op
 SPECIFICATION
  INPUT
   fire_cmd2_str: target_data
  OUTPUT
   fire_cmd3_str: target_data
  MAXIMUM EXECUTION TIME 50 MS
 END
```

```
 IMPLEMENTATION ADA ctoc_op
 END

OPERATOR grnd_stat_mod_op
 SPECIFICATION
  INPUT
   target_array2_str: jstars_array
  OUTPUT
   target_array3_str: grnd_stat_mod_array
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA grnd_stat_mod_op
 END

OPERATOR gui_input_event_monitor
 SPECIFICATION
  MAXIMUM EXECUTION TIME 200 MS
 END

 IMPLEMENTATION ADA gui_input_event_monitor
 END

OPERATOR jstars_op
 SPECIFICATION
  INPUT
   emission_str: target_emitter_array,
   gui_in_str: my_unit
  OUTPUT
   target_array1_str: jstars_array
  MAXIMUM EXECUTION TIME 500 MS
 END

 IMPLEMENTATION ADA jstars_op
 END

OPERATOR lan1_link_op
 SPECIFICATION
  INPUT
   target_array3_str: grnd_stat_mod_array
  OUTPUT
   target_array4_str: grnd_stat_mod_array
  MAXIMUM EXECUTION TIME 50 MS
 END
```

```
 IMPLEMENTATION ADA lan1_link_op
 END

OPERATOR lan2_link_op
 SPECIFICATION
  INPUT
   fire_cmd1_str: target_data
  OUTPUT
   fire_cmd2_str: target_data
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA lan2_link_op
 END

OPERATOR scdl_link_op
 SPECIFICATION
  INPUT
   target_array1_str: jstars_array
  OUTPUT
   target_array2_str: jstars_array
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA scdl_link_op
 END

OPERATOR shooter_op
 SPECIFICATION
  INPUT
   fire_cmd4_str: target_data
  OUTPUT
   gui_out_str: target_data
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA shooter_op
 END

OPERATOR target_emitter_op
 SPECIFICATION
  INPUT
   gui_in_str: my_unit
  OUTPUT
   emission_str: target_emitter_array
```

```
  MAXIMUM EXECUTION TIME 500 MS
 END

 IMPLEMENTATION ADA target_emitter_op
 END
```

asas_op's ancestor chain: atacms->command_station_op
choose_inputs's ancestor chain: atacms->gui_in
cmds_out's ancestor chain: atacms->gui_out
cnr_link_op's ancestor chain: atacms
ctoc_op's ancestor chain: atacms->command_station_op
grnd_stat_mod_op's ancestor chain: atacms->command_station_op
gui_input_event_monitor's ancestor chain: atacms->gui_in
jstars_op's ancestor chain: atacms
lan1_link_op's ancestor chain: atacms->command_station_op
lan2_link_op's ancestor chain: atacms->command_station_op
scdl_link_op's ancestor chain: atacms
shooter_op's ancestor chain: atacms
target_emitter_op's ancestor chain: atacms

Test-Case: test_conflict

This test-case demonstrates *ancestor chain* conflict reporting and resolution as well as showing that a very reasonable decomposition structure can be recovered in the case of decomposition structure merge conflicts.

For this test-case, I created atacms.A.Conflict.psdl from atacms.A.psdl (used in test_dg_1) by renaming composite operator gui_out to gui_out_conflict. I used atacms.BASE.psdl (used in test_dg_1) for atacms.Base.Conflict.psdl, and atacms.psdl for atacms.B.Conflict.pdsl.

Test-Driver: test_conflict

```
with TEXT_IO; use TEXT_IO;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_io; use psdl_io;
with extended_ancestor_pkg; use extended_ancestor_pkg;
with ancestor_chains_pkg; use ancestor_chains_pkg;
with decompose_graph_pkg; use decompose_graph_pkg;
procedure test_conflict is
        TESTFILE: FILE_TYPE;
        NEW_PSDL, A_PSDL, BASE_PSDL, B_PSDL, MERGE: psdl_program;
    root_op: psdl_id;
```

```
        ancestors: ancestor_chains;

      MERGE_CHAIN: extended_ancestor := null_ancestor;

begin
        OPEN(TESTFILE,IN_FILE,"atacms_ex.psdl");
        assign(MERGE,empty_psdl_program);

        put_line("getting MERGE prototype file!");
        get(TESTFILE,MERGE);

        CLOSE(TESTFILE);
        -- put(MERGE);

        OPEN(TESTFILE,IN_FILE,"atacms.A.Conflict.psdl");
        assign(A_PSDL,empty_psdl_program);

        put_line("getting change A prototype file!");
        get(TESTFILE,A_PSDL);

        CLOSE(TESTFILE);
        -- put(A_PSDL);

        OPEN(TESTFILE,IN_FILE,"atacms.Base.Conflict.psdl");
        assign(BASE_PSDL,empty_psdl_program);

        put_line("getting change BASE prototype file!");
        get(TESTFILE,BASE_PSDL);

        CLOSE(TESTFILE);
        -- put(BASE_PSDL);

        OPEN(TESTFILE,IN_FILE,"atacms.B.Conflict.psdl");
        assign(B_PSDL,empty_psdl_program);

        put_line("getting change B prototype file!");
        get(TESTFILE,B_PSDL);

        CLOSE(TESTFILE);
        -- put(B_PSDL);

        decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE, NEW_PSDL);

        -- need the root operator for find_ancestor_chain.
```

```
    root_op := find_root(NEW_PSDL);
    put_line(convert(root_op));

    put(NEW_PSDL);
    ancestor_chains_map_inst_pkg.assign(ancestors, empty_ancestor_chains);
  for id: psdl_id, c : psdl_component in psdl_program_map_pkg.scan(NEW_PSDL)
    loop
            if component_category(c) = psdl_operator then
                    if component_granularity(c) = atomic then
                    MERGE_CHAIN := find_ancestor_chain(id, root_op,
                                                                   NEW_PSDL);
                            ancestor_chains_map_inst_pkg.bind(id, MERGE_CHAIN,
                                                                        ancestors);
                    end if;
            end if;
  end loop;

    put_ancestor_chains(ancestors);
    ancestor_chains_map_inst_pkg.recycle(ancestors);
end test_conflict;
```

Test-Output: test_conflict

```
getting MERGE prototype file!
getting change A prototype file!
getting change BASE prototype file!
getting change B prototype file!
ONE OR MORE CONFLICTS IN ANCESTOR CHAIN RECOVERY FOR: cmds_out
<atacms->gui_out_conflict>
[<EMPTY CHAIN>]
<atacms->gui_out> =
<atacms->gui_out> U
<atacms->gui_out_conflict>=
(***conflict***) =
<atacms(gui_out U gui_out_conflict)>

asas_op's ancestor chain: atacms->command_station_op
choose_inputs's ancestor chain: atacms->gui_in
cnr_link_op's ancestor chain: atacms
ctoc_op's ancestor chain: atacms->command_station_op
grnd_stat_mod_op's ancestor chain: atacms->command_station_op
gui_input_event_monitor's ancestor chain: atacms->gui_in
jstars_op's ancestor chain: atacms
lan1_link_op's ancestor chain: atacms->command_station_op
lan2_link_op's ancestor chain: atacms->command_station_op
```

```
scdl_link_op's ancestor chain: atacms
shooter_op's ancestor chain: atacms
target_emitter_op's ancestor chain: atacms
cmds_out's ancestor chain: atacms
atacms
OPERATOR atacms
 SPECIFICATION
  STATES gui_in_str: my_unit INITIALLY pause
 END

 IMPLEMENTATION
  GRAPH
   VERTEX command_station_op
   VERTEX gui_in
   VERTEX cnr_link_op: 50 MS
   VERTEX jstars_op: 500 MS
   VERTEX scdl_link_op: 50 MS
   VERTEX shooter_op: 50 MS
   VERTEX target_emitter_op: 500 MS
   VERTEX cmds_out

   EDGE fire_cmd4_str cnr_link_op -> shooter_op
   EDGE emission_str target_emitter_op -> jstars_op
   EDGE target_array1_str jstars_op -> scdl_link_op
   EDGE gui_out_str shooter_op -> cmds_out
   EDGE gui_in_str gui_in -> command_station_op
   EDGE fire_cmd3_str command_station_op -> cnr_link_op
   EDGE gui_in_str gui_in -> jstars_op
   EDGE target_array2_str scdl_link_op -> command_station_op
   EDGE gui_in_str gui_in -> target_emitter_op

  DATA STREAM
   fire_cmd4_str: target_data,
   fire_cmd3_str: target_data,
   emission_str: target_emitter_array,
   target_array1_str: jstars_array,
   target_array2_str: jstars_array,
   gui_out_str: target_data
  CONTROL CONSTRAINTS
   OPERATOR command_station_op
   OPERATOR gui_in
   OPERATOR cnr_link_op
    TRIGGERED BY SOME fire_cmd3_str
   OPERATOR jstars_op
    TRIGGERED IF (gui_in_str /= my_unit.pause)
```

```
    PERIOD 8000 MS
   OPERATOR scdl_link_op
    TRIGGERED BY SOME target_array1_str
   OPERATOR shooter_op
    TRIGGERED BY SOME fire_cmd4_str
   OPERATOR target_emitter_op
    TRIGGERED IF (gui_in_str /= my_unit.pause)
    PERIOD 16000 MS
   OPERATOR cmds_out
    TRIGGERED BY SOME gui_out_str
 END

OPERATOR command_station_op
 SPECIFICATION
  INPUT
   gui_in_str: my_unit,
   target_array2_str: jstars_array
  OUTPUT
   fire_cmd3_str: target_data
 END

 IMPLEMENTATION
  GRAPH
   VERTEX asas_op: 200 MS
   VERTEX ctoc_op: 50 MS
   VERTEX grnd_stat_mod_op: 50 MS
   VERTEX lan1_link_op: 50 MS
   VERTEX lan2_link_op: 50 MS

   EDGE fire_cmd1_str asas_op -> lan2_link_op
   EDGE target_array4_str lan1_link_op -> asas_op
   EDGE fire_cmd2_str lan2_link_op -> ctoc_op
   EDGE target_array3_str grnd_stat_mod_op -> lan1_link_op
   EDGE gui_in_str EXTERNAL -> asas_op
   EDGE target_array2_str : 5000 MS EXTERNAL -> grnd_stat_mod_op
   EDGE fire_cmd3_str ctoc_op -> EXTERNAL

  DATA STREAM
   fire_cmd1_str: target_data,
   target_array4_str: grnd_stat_mod_array,
   fire_cmd2_str: target_data,
   target_array3_str: grnd_stat_mod_array
  CONTROL CONSTRAINTS
   OPERATOR asas_op
    TRIGGERED IF (gui_in_str /= my_unit.pause)
```

```
    PERIOD 4000 MS
   OPERATOR ctoc_op
    TRIGGERED BY SOME fire_cmd2_str
   OPERATOR grnd_stat_mod_op
    TRIGGERED BY SOME target_array2_str
   OPERATOR lan1_link_op
    TRIGGERED BY SOME target_array3_str
   OPERATOR lan2_link_op
    TRIGGERED BY SOME fire_cmd1_str
 END

OPERATOR asas_op
 SPECIFICATION
  INPUT
   gui_in_str: my_unit,
   target_array4_str: grnd_stat_mod_array
  OUTPUT
   fire_cmd1_str: target_data
  MAXIMUM EXECUTION TIME 200 MS
 END

 IMPLEMENTATION ADA asas_op
 END

OPERATOR gui_in
 SPECIFICATION
  OUTPUT
   gui_in_str: my_unit
 END

 IMPLEMENTATION
  GRAPH
   VERTEX choose_inputs: 200 MS
   VERTEX gui_input_event_monitor: 200 MS

   EDGE gui_in_str choose_inputs -> EXTERNAL

  CONTROL CONSTRAINTS
   OPERATOR choose_inputs
    PERIOD 2000 MS
   OPERATOR gui_input_event_monitor
 END

OPERATOR choose_inputs
 SPECIFICATION
```

```
   OUTPUT
    gui_in_str: my_unit
   MAXIMUM EXECUTION TIME 200 MS
  END

  IMPLEMENTATION ADA choose_inputs
  END

OPERATOR cnr_link_op
  SPECIFICATION
   INPUT
    fire_cmd3_str: target_data
   OUTPUT
    fire_cmd4_str: target_data
   MAXIMUM EXECUTION TIME 50 MS
  END

  IMPLEMENTATION ADA cnr_link_op
  END

OPERATOR ctoc_op
  SPECIFICATION
   INPUT
    fire_cmd2_str: target_data
   OUTPUT
    fire_cmd3_str: target_data
   MAXIMUM EXECUTION TIME 50 MS
  END

  IMPLEMENTATION ADA ctoc_op
  END

OPERATOR grnd_stat_mod_op
  SPECIFICATION
   INPUT
    target_array2_str: jstars_array
   OUTPUT
    target_array3_str: grnd_stat_mod_array
   MAXIMUM EXECUTION TIME 50 MS
  END

  IMPLEMENTATION ADA grnd_stat_mod_op
  END

OPERATOR gui_input_event_monitor
```

```
 SPECIFICATION
  MAXIMUM EXECUTION TIME 200 MS
 END

 IMPLEMENTATION ADA gui_input_event_monitor
 END

OPERATOR jstars_op
 SPECIFICATION
  INPUT
   emission_str: target_emitter_array,
   gui_in_str: my_unit
  OUTPUT
   target_array1_str: jstars_array
  MAXIMUM EXECUTION TIME 500 MS
 END

 IMPLEMENTATION ADA jstars_op
 END

OPERATOR lan1_link_op
 SPECIFICATION
  INPUT
   target_array3_str: grnd_stat_mod_array
  OUTPUT
   target_array4_str: grnd_stat_mod_array
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA lan1_link_op
 END

OPERATOR lan2_link_op
 SPECIFICATION
  INPUT
   fire_cmd1_str: target_data
  OUTPUT
   fire_cmd2_str: target_data
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA lan2_link_op
 END

OPERATOR scdl_link_op
```

```
 SPECIFICATION
  INPUT
   target_array1_str: jstars_array
  OUTPUT
   target_array2_str: jstars_array
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA scdl_link_op
 END

OPERATOR shooter_op
 SPECIFICATION
  INPUT
   fire_cmd4_str: target_data
  OUTPUT
   gui_out_str: target_data
  MAXIMUM EXECUTION TIME 50 MS
 END

 IMPLEMENTATION ADA shooter_op
 END

OPERATOR target_emitter_op
 SPECIFICATION
  INPUT
   gui_in_str: my_unit
  OUTPUT
   emission_str: target_emitter_array
  MAXIMUM EXECUTION TIME 500 MS
 END

 IMPLEMENTATION ADA target_emitter_op
 END

OPERATOR cmds_out
 SPECIFICATION
  INPUT
   gui_out_str: target_data
 END

 IMPLEMENTATION ADA cmds_out
 END
```

asas_op's ancestor chain: atacms->command_station_op

choose_inputs's ancestor chain: atacms->gui_in
cnr_link_op's ancestor chain: atacms
ctoc_op's ancestor chain: atacms->command_station_op
grnd_stat_mod_op's ancestor chain: atacms->command_station_op
gui_input_event_monitor's ancestor chain: atacms->gui_in
jstars_op's ancestor chain: atacms
lan1_link_op's ancestor chain: atacms->command_station_op
lan2_link_op's ancestor chain: atacms->command_station_op
scdl_link_op's ancestor chain: atacms
shooter_op's ancestor chain: atacms
target_emitter_op's ancestor chain: atacms
cmds_out's ancestor chain: atacms

Test-Case: test_dg_2

This test-case is similar to test_dg_1 except for the prototype used, c3I_system.psdl, is roughly twice as large as atacms.psdl. This test-case also demonstrates saving a reconstructed prototype to file -- c3i_system.NEW.psdl.

For MERGE, I used an expanded version of c3i_system, c3i_system.ex.psdl, and edited c3i_system.psdl to create A, BASE, and B. c3i_system.A.psdl has composite operator sensor_interface (and all associated streams and vertices) removed. c3I_system.B.psdl has atomic operators weapons_interface, weapons_system, and emergency_status_screen removed. c3i_system.Base.psdl has composite operator sensor_interface (and all associated streams and vertices) removed as well as atomic operators weapons_interface, weapons_system, and emergency_status_screen.

Test-Driver: test_dg_2

```
with TEXT_IO; use TEXT_IO;
with psdl_component_pkg; use psdl_component_pkg;
with psdl_concrete_type_pkg; use psdl_concrete_type_pkg;
with psdl_program_pkg; use psdl_program_pkg;
with psdl_io; use psdl_io;
with extended_ancestor_pkg; use extended_ancestor_pkg;
with ancestor_chains_pkg; use ancestor_chains_pkg;
with decompose_graph_pkg; use decompose_graph_pkg;
procedure test_dg_2 is
        TESTFILE: FILE_TYPE;
        NEW_PSDL, A_PSDL, BASE_PSDL, B_PSDL, MERGE: psdl_program;
    root_op: psdl_id;

        ancestors: ancestor_chains;

    MERGE_CHAIN: extended_ancestor := null_ancestor;
```

```
begin
    OPEN(TESTFILE,IN_FILE,"c3i_system.ex.psdl");
    assign(MERGE,empty_psdl_program);

    put_line("getting MERGE prototype file!");
    get(TESTFILE,MERGE);

    CLOSE(TESTFILE);
    -- put(MERGE);

    OPEN(TESTFILE,IN_FILE,"c3i_system.A.psdl");
    assign(A_PSDL,empty_psdl_program);

    put_line("getting change A prototype file!");
    get(TESTFILE,A_PSDL);

    CLOSE(TESTFILE);
    -- put(A_PSDL);

    OPEN(TESTFILE,IN_FILE,"c3i_system.Base.psdl");
    assign(BASE_PSDL,empty_psdl_program);

    put_line("getting change BASE prototype file!");
    get(TESTFILE,BASE_PSDL);

    CLOSE(TESTFILE);
    -- put(BASE_PSDL);

    OPEN(TESTFILE,IN_FILE,"c3i_system.B.psdl");
    assign(B_PSDL,empty_psdl_program);

    put_line("getting change B prototype file!");
    get(TESTFILE,B_PSDL);

    CLOSE(TESTFILE);
    -- put(B_PSDL);

    decompose_graph(A_PSDL, BASE_PSDL, B_PSDL, MERGE, NEW_PSDL);

    -- need the root operator for find_ancestor_chain.
    root_op := find_root(NEW_PSDL);
    put("PROTOTYPE ROOT OPERATOR NAME: ");
    put_line(convert(root_op));
```

```
    put(NEW_PSDL);
    ancestor_chains_map_inst_pkg.assign(ancestors, empty_ancestor_chains);
  for id: psdl_id, c : psdl_component in psdl_program_map_pkg.scan(NEW_PSDL)
    loop
          if component_category(c) = psdl_operator then
                  if component_granularity(c) = atomic then
                  MERGE_CHAIN := find_ancestor_chain(id, root_op,
                                                              NEW_PSDL);
                          ancestor_chains_map_inst_pkg.bind(id, MERGE_CHAIN,
                                                                   ancestors);
                  end if;
          end if;
  end loop;

    put_ancestor_chains(ancestors);
    ancestor_chains_map_inst_pkg.recycle(ancestors);

    OPEN(TESTFILE,OUT_FILE,"c3i_system.NEW.psdl");
    put(TESTFILE, NEW_PSDL);
    CLOSE(TESTFILE);
end test_dg_2;
```

Test-Output: test_dg_2

```
getting MERGE prototype file!
getting change A prototype file!
getting change BASE prototype file!
getting change B prototype file!
convert_to_text_file's ancestor chain: c3i_system->comms_interface
decide_for_archiving's ancestor chain: c3i_system->comms_interface
extract_tracks's ancestor chain: c3i_system->comms_interface
forward_for_transmission's ancestor chain: c3i_system->comms_interface
make_routing's ancestor chain: c3i_system->comms_interface
parse_input_file's ancestor chain: c3i_system->comms_interface
prepare_periodic_report's ancestor chain: c3i_system->comms_interface
comms_links's ancestor chain: c3i_system
navigation_system's ancestor chain: c3i_system
analyze_sensor_data's ancestor chain: c3i_system->sensor_interface
prepare_sensor_track's ancestor chain: c3i_system->sensor_interface
sensors's ancestor chain: c3i_system
add_comms_track's ancestor chain: c3i_system->track_database_manager
add_sensor_track's ancestor chain: c3i_system->track_database_manager
filter_comms_tracks's ancestor chain: c3i_system->track_database_manager
filter_sensor_tracks's ancestor chain: c3i_system->track_database_manager
monitor_ownship_position's ancestor chain: c3i_system->track_database_manager
```

```
display_tracks's ancestor chain: c3i_system->user_interface
emergency_status_screen's ancestor chain: c3i_system->user_interface
get_user_inputs's ancestor chain: c3i_system->user_interface
manage_user_interface's ancestor chain: c3i_system->user_interface
message_arrival_panel's ancestor chain: c3i_system->user_interface
message_editor's ancestor chain: c3i_system->user_interface
status_screen's ancestor chain: c3i_system->user_interface
weapons_interface's ancestor chain: c3i_system
weapons_systems's ancestor chain: c3i_system
PROTOTYPE ROOT OPERATOR NAME: c3i_system
OPERATOR c3i_system
 SPECIFICATION
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION
  GRAPH
   VERTEX comms_interface
   VERTEX comms_links: 1200 MS
   VERTEX navigation_system: 800 MS
   VERTEX sensor_interface
   VERTEX sensors: 800 MS
   VERTEX track_database_manager
   VERTEX user_interface
   VERTEX weapons_interface: 500 MS
   VERTEX weapons_systems: 500 MS

   EDGE weapon_status_data weapons_systems -> weapons_interface
   EDGE comms_email comms_interface -> user_interface
   EDGE tdd_archive_setup user_interface -> comms_interface
   EDGE comms_add_track comms_interface -> track_database_manager
   EDGE tcd_emission_control user_interface -> comms_interface
   EDGE tcd_transmit_command user_interface -> comms_interface
   EDGE tcd_network_setup user_interface -> comms_interface
   EDGE initiate_trans user_interface -> comms_interface
   EDGE terminate_trans user_interface -> comms_interface
   EDGE sensor_add_track sensor_interface -> track_database_manager
   EDGE tdd_filter user_interface -> track_database_manager
   EDGE out_tracks track_database_manager -> user_interface
   EDGE input_link_message comms_links -> comms_interface
   EDGE position_data navigation_system -> track_database_manager
   EDGE position_data navigation_system -> sensor_interface
   EDGE sensor_data sensors -> sensor_interface
   EDGE weapons_emrep weapons_interface -> user_interface
   EDGE weapons_statrep weapons_interface -> user_interface
```

```
  DATA STREAM
    input_link_message: filename,
    position_data: ownship_navigation_info,
    sensor_data: sensor_record,
    weapon_status_data: weapon_status,
    weapons_emrep: weapon_status_report,
    weapons_statrep: weapon_status_report,
    comms_email: filename,
    tdd_archive_setup: archive_setup,
    comms_add_track: add_track_tuple,
    tcd_emission_control: emissions_control_command,
    tcd_transmit_command: type1,
    tcd_network_setup: network_setup,
    initiate_trans: initiate_transmission_sequence,
    terminate_trans: boolean,
    sensor_add_track: add_track_tuple,
    tdd_filter: set_track_filter,
    out_tracks: track_tuple
  CONTROL CONSTRAINTS
    OPERATOR comms_interface
    OPERATOR comms_links
      PERIOD 50000 MS
    OPERATOR navigation_system
      PERIOD 50000 MS
    OPERATOR sensor_interface
    OPERATOR sensors
      PERIOD 50000 MS
    OPERATOR track_database_manager
    OPERATOR user_interface
    OPERATOR weapons_interface
      TRIGGERED BY SOME weapon_status_data
      OUTPUT weapons_emrep IF (((weapon_status_data.status = DAMAGED) OR
(weapon_status_data.status = SERVICE_REQUIRED)) OR (weapon_status_data.status =
OUT_OF_AMMUNITION))
    OPERATOR weapons_systems
      PERIOD 50000 MS
 END

OPERATOR comms_interface
 SPECIFICATION
  INPUT
    tdd_archive_setup: archive_setup,
    tcd_emission_control: emissions_control_command,
    tcd_transmit_command: transmit_command,
```

```
   tcd_network_setup: network_setup,
   input_link_message: filename,
   initiate_trans: initiate_transmission_sequence,
   terminate_trans: boolean
  OUTPUT
   comms_email: filename,
   comms_add_track: add_track_tuple
  DESCRIPTION { <text> }
END

IMPLEMENTATION
 GRAPH
  VERTEX convert_to_text_file: 800 MS
  VERTEX decide_for_archiving: 500 MS
  VERTEX extract_tracks: 500 MS
  VERTEX forward_for_transmission: 500 MS
  VERTEX make_routing: 500 MS
  VERTEX parse_input_file: 500 MS
  VERTEX prepare_periodic_report: 800 MS

  EDGE output_messages forward_for_transmission -> convert_to_text_file
  EDGE input_text_record parse_input_file -> decide_for_archiving
  EDGE comms_text_file decide_for_archiving -> extract_tracks
  EDGE transmission_message make_routing -> forward_for_transmission
  EDGE tcd_transmit_command prepare_periodic_report -> make_routing
  EDGE tdd_archive_setup EXTERNAL -> decide_for_archiving
  EDGE tcd_emission_control EXTERNAL -> forward_for_transmission
  EDGE tcd_transmit_command EXTERNAL -> make_routing
  EDGE tcd_network_setup EXTERNAL -> make_routing
  EDGE input_link_message EXTERNAL -> parse_input_file
  EDGE initiate_trans EXTERNAL -> prepare_periodic_report
  EDGE terminate_trans EXTERNAL -> prepare_periodic_report
  EDGE comms_email decide_for_archiving -> EXTERNAL
  EDGE comms_add_track extract_tracks -> EXTERNAL

 DATA STREAM
  output_messages: message_list,
  input_text_record: text_record,
  comms_text_file: text_record,
  transmission_message: transmission_command
 CONTROL CONSTRAINTS
  OPERATOR convert_to_text_file
   TRIGGERED BY SOME output_messages
  OPERATOR decide_for_archiving
   TRIGGERED BY SOME input_text_record
```

```
    OUTPUT comms_text_file IF comms_text_file.archive
    OUTPUT comms_email IF NOT(comms_text_file.is_track)
   OPERATOR extract_tracks
    TRIGGERED IF comms_text_file.is_track
   OPERATOR forward_for_transmission
    TRIGGERED BY SOME transmission_message
    OUTPUT output_messages IF (tcd_emission_control = UNRESTRICTED)
   OPERATOR make_routing
    TRIGGERED BY SOME tcd_transmit_command
   OPERATOR parse_input_file
    TRIGGERED BY SOME input_link_message
   OPERATOR prepare_periodic_report
    TRIGGERED IF NOT(terminate_trans)
    PERIOD 50000 MS
 END

OPERATOR convert_to_text_file
 SPECIFICATION
  INPUT
   output_messages: message_list
  MAXIMUM EXECUTION TIME 800 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA convert_to_text_file
 END

OPERATOR decide_for_archiving
 SPECIFICATION
  INPUT
   input_text_record: text_record,
   tdd_archive_setup: archive_setup
  OUTPUT
   comms_text_file: text_record,
   comms_email: filename
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA decide_for_archiving
 END

OPERATOR extract_tracks
 SPECIFICATION
  INPUT
```

```
    comms_text_file: text_record
   OUTPUT
    comms_add_track: add_track_tuple
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA extract_tracks
 END

OPERATOR forward_for_transmission
 SPECIFICATION
  INPUT
   transmission_message: transmission_command,
   tcd_emission_control: emissions_control_command
  OUTPUT
   output_messages: message_list
  STATES waiting_messages: message_list INITIALLY null
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA forward_for_transmission
 END

OPERATOR make_routing
 SPECIFICATION
  INPUT
   tcd_transmit_command: transmit_command,
   tcd_network_setup: network_setup
  OUTPUT
   transmission_message: transmission_command
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA make_routing
 END

OPERATOR parse_input_file
 SPECIFICATION
  INPUT
   input_link_message: filename
  OUTPUT
   input_text_record: text_record
```

```
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA parse_input_file
 END

OPERATOR prepare_periodic_report
 SPECIFICATION
  INPUT
   initiate_trans: initiate_transmission_sequence,
   terminate_trans: boolean
  OUTPUT
   tcd_transmit_command: transmit_command
  MAXIMUM EXECUTION TIME 800 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA prepare_periodic_report
 END

OPERATOR comms_links
 SPECIFICATION
  OUTPUT
   input_link_message: filename
  MAXIMUM EXECUTION TIME 1200 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA comms_links
 END

OPERATOR navigation_system
 SPECIFICATION
  OUTPUT
   position_data: ownship_navigation_info
  MAXIMUM EXECUTION TIME 800 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA navigation_system
 END

OPERATOR sensor_interface
 SPECIFICATION
```

```
  INPUT
    sensor_data: sensor_record,
    position_data: ownship_navigation_info
  OUTPUT
    sensor_add_track: add_track_tuple
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION
  GRAPH
    VERTEX analyze_sensor_data: 500 MS
    VERTEX prepare_sensor_track: 500 MS

    EDGE sensor_contact_data analyze_sensor_data -> prepare_sensor_track
    EDGE sensor_data EXTERNAL -> analyze_sensor_data
    EDGE position_data EXTERNAL -> prepare_sensor_track
    EDGE sensor_add_track prepare_sensor_track -> EXTERNAL

  DATA STREAM
    sensor_contact_data: local_track_info
  CONTROL CONSTRAINTS
    OPERATOR analyze_sensor_data
      TRIGGERED BY SOME sensor_data
    OPERATOR prepare_sensor_track
      TRIGGERED BY ALL sensor_contact_data, position_data
 END

OPERATOR analyze_sensor_data
 SPECIFICATION
  INPUT
    sensor_data: sensor_record
  OUTPUT
    sensor_contact_data: local_track_info
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA analyze_sensor_data
 END

OPERATOR prepare_sensor_track
 SPECIFICATION
  INPUT
    sensor_contact_data: local_track_info,
    position_data: ownship_navigation_info
```

```
  OUTPUT
   sensor_add_track: add_track_tuple
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA prepare_sensor_track
 END

OPERATOR sensors
 SPECIFICATION
  OUTPUT
   sensor_data: sensor_record
  MAXIMUM EXECUTION TIME 800 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA sensors
 END

OPERATOR track_database_manager
 SPECIFICATION
  INPUT
   tdd_filter: set_track_filter,
   comms_add_track: add_track_tuple,
   sensor_add_track: add_track_tuple,
   position_data: ownship_navigation_info
  OUTPUT
   out_tracks: track_tuple
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION
  GRAPH
   VERTEX add_comms_track: 500 MS
   VERTEX add_sensor_track: 500 MS
   VERTEX filter_comms_tracks: 500 MS
   VERTEX filter_sensor_tracks: 500 MS
   VERTEX monitor_ownship_position: 500 MS

   EDGE filtered_comms_track filter_comms_tracks -> add_comms_track
   EDGE filtered_sensor_track filter_sensor_tracks -> add_sensor_track
   EDGE tdd_filter EXTERNAL -> add_comms_track
   EDGE tdd_filter EXTERNAL -> add_sensor_track
   EDGE tdd_filter EXTERNAL -> filter_comms_tracks
```

```
    EDGE tdd_filter EXTERNAL -> filter_sensor_tracks
    EDGE comms_add_track EXTERNAL -> filter_comms_tracks
    EDGE sensor_add_track EXTERNAL -> filter_sensor_tracks
    EDGE position_data EXTERNAL -> monitor_ownship_position
    EDGE out_tracks add_comms_track -> EXTERNAL
    EDGE out_tracks add_sensor_track -> EXTERNAL
    EDGE out_tracks monitor_ownship_position -> EXTERNAL

   DATA STREAM
    filtered_comms_track: add_track_tuple,
    filtered_sensor_track: add_track_tuple
   CONTROL CONSTRAINTS
    OPERATOR add_comms_track
     TRIGGERED BY SOME filtered_comms_track
    OPERATOR add_sensor_track
     TRIGGERED BY SOME filtered_sensor_track
    OPERATOR filter_comms_tracks
     TRIGGERED BY SOME comms_add_track
    OPERATOR filter_sensor_tracks
     TRIGGERED BY SOME sensor_add_track
    OPERATOR monitor_ownship_position
     TRIGGERED BY SOME position_data
  END

OPERATOR add_comms_track
 SPECIFICATION
  INPUT
   filtered_comms_track: add_track_tuple,
   tdd_filter: set_track_filter
  OUTPUT
   out_tracks: track_tuple
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA add_comms_track
 END

OPERATOR add_sensor_track
 SPECIFICATION
  INPUT
   filtered_sensor_track: add_track_tuple,
   tdd_filter: set_track_filter
  OUTPUT
   out_tracks: track_tuple
```

```
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA add_sensor_track
 END

OPERATOR filter_comms_tracks
 SPECIFICATION
  INPUT
   comms_add_track: add_track_tuple,
   tdd_filter: set_track_filter
  OUTPUT
   filtered_comms_track: add_track_tuple
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA filter_comms_tracks
 END

OPERATOR filter_sensor_tracks
 SPECIFICATION
  INPUT
   sensor_add_track: add_track_tuple,
   tdd_filter: set_track_filter
  OUTPUT
   filtered_sensor_track: add_track_tuple
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA filter_sensor_tracks
 END

OPERATOR monitor_ownship_position
 SPECIFICATION
  INPUT
   position_data: ownship_navigation_info
  OUTPUT
   out_tracks: track_tuple
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END
```

```
IMPLEMENTATION ADA monitor_ownship_position
END

OPERATOR user_interface
 SPECIFICATION
  INPUT
   out_tracks: track_tuple,
   weapons_emrep: weapon_status_report,
   comms_email: filename,
   weapons_statrep: weapon_status_report
  OUTPUT
   tdd_archive_setup: archive_setup,
   initiate_trans: initiate_transmission_sequence,
   terminate_trans: boolean,
   tcd_network_setup: network_setup,
   tcd_emission_control: emissions_control_command,
   tdd_filter: set_track_filter,
   tcd_transmit_command: transmit_command
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION
  GRAPH
   VERTEX display_tracks
   VERTEX emergency_status_screen
   VERTEX get_user_inputs
   VERTEX manage_user_interface
   VERTEX message_arrival_panel
   VERTEX message_editor
   VERTEX status_screen

   EDGE td_track_request get_user_inputs -> display_tracks
   EDGE tcd_status_query get_user_inputs -> status_screen
   EDGE editor_selected get_user_inputs -> message_editor
   EDGE out_tracks EXTERNAL -> display_tracks
   EDGE weapons_emrep EXTERNAL -> emergency_status_screen
   EDGE comms_email EXTERNAL -> message_arrival_panel
   EDGE weapons_statrep EXTERNAL -> status_screen
   EDGE tdd_archive_setup get_user_inputs -> EXTERNAL
   EDGE initiate_trans get_user_inputs -> EXTERNAL
   EDGE terminate_trans get_user_inputs -> EXTERNAL
   EDGE tcd_network_setup get_user_inputs -> EXTERNAL
   EDGE tcd_emission_control get_user_inputs -> EXTERNAL
   EDGE tdd_filter get_user_inputs -> EXTERNAL
   EDGE tcd_transmit_command message_editor -> EXTERNAL
```

```
  DATA STREAM
    td_track_request: database_request,
    tcd_status_query: boolean,
    editor_selected: boolean
  CONTROL CONSTRAINTS
    OPERATOR display_tracks
      TRIGGERED BY SOME out_tracks
    OPERATOR emergency_status_screen
      TRIGGERED BY SOME weapons_emrep
    OPERATOR get_user_inputs
    OPERATOR manage_user_interface
    OPERATOR message_arrival_panel
      TRIGGERED BY SOME comms_email
    OPERATOR message_editor
      TRIGGERED IF editor_selected
    OPERATOR status_screen
      TRIGGERED IF tcd_status_query
 END

OPERATOR display_tracks
 SPECIFICATION
  INPUT
    out_tracks: track_tuple,
    td_track_request: database_request
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA display_tracks
 END

OPERATOR emergency_status_screen
 SPECIFICATION
  INPUT
    weapons_emrep: weapon_status_report
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA emergency_status_screen
 END

OPERATOR get_user_inputs
 SPECIFICATION
  OUTPUT
    tdd_archive_setup: archive_setup,
```

```
      tdd_filter: set_track_filter,
      td_track_request: database_request,
      tcd_status_query: boolean,
      tcd_network_setup: network_setup,
      tcd_emission_control: emissions_control_command,
      editor_selected: boolean,
      initiate_trans: initiate_transmission_sequence,
      terminate_trans: boolean
    DESCRIPTION { <text> }
  END

  IMPLEMENTATION ADA get_user_inputs
  END

OPERATOR manage_user_interface
  SPECIFICATION
    DESCRIPTION { <text> }
  END

  IMPLEMENTATION ADA manage_user_interface
  END

OPERATOR message_arrival_panel
  SPECIFICATION
    INPUT
      comms_email: filename
    DESCRIPTION { <text> }
  END

  IMPLEMENTATION ADA message_arrival_panel
  END

OPERATOR message_editor
  SPECIFICATION
    INPUT
      editor_selected: boolean
    OUTPUT
      tcd_transmit_command: transmit_command
    DESCRIPTION { <text> }
  END

  IMPLEMENTATION ADA message_editor
  END

OPERATOR status_screen
```

```
 SPECIFICATION
  INPUT
   weapons_statrep: weapon_status_report,
   tcd_status_query: boolean
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA status_screen
 END

OPERATOR weapons_interface
 SPECIFICATION
  INPUT
   weapon_status_data: weapon_status
  OUTPUT
   weapons_emrep: weapon_status_report,
   weapons_statrep: weapon_status_report
  STATES ciws_status: weapon_status_type INITIALLY READY
  STATES gun_status: weapon_status_type INITIALLY READY
  STATES tws_status: weapon_status_type INITIALLY READY
  STATES mk_48_status: weapon_status_type INITIALLY READY
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA weapons_interface
 END

OPERATOR weapons_systems
 SPECIFICATION
  OUTPUT
   weapon_status_data: weapon_status
  MAXIMUM EXECUTION TIME 500 MS
  DESCRIPTION { <text> }
 END

 IMPLEMENTATION ADA weapons_systems
 END

convert_to_text_file's ancestor chain: c3i_system->comms_interface
decide_for_archiving's ancestor chain: c3i_system->comms_interface
extract_tracks's ancestor chain: c3i_system->comms_interface
forward_for_transmission's ancestor chain: c3i_system->comms_interface
make_routing's ancestor chain: c3i_system->comms_interface
parse_input_file's ancestor chain: c3i_system->comms_interface
```

prepare_periodic_report's ancestor chain: c3i_system->comms_interface
comms_links's ancestor chain: c3i_system
navigation_system's ancestor chain: c3i_system
analyze_sensor_data's ancestor chain: c3i_system->sensor_interface
prepare_sensor_track's ancestor chain: c3i_system->sensor_interface
sensors's ancestor chain: c3i_system
add_comms_track's ancestor chain: c3i_system->track_database_manager
add_sensor_track's ancestor chain: c3i_system->track_database_manager
filter_comms_tracks's ancestor chain: c3i_system->track_database_manager
filter_sensor_tracks's ancestor chain: c3i_system->track_database_manager
monitor_ownship_position's ancestor chain: c3i_system->track_database_manager
display_tracks's ancestor chain: c3i_system->user_interface
emergency_status_screen's ancestor chain: c3i_system->user_interface
get_user_inputs's ancestor chain: c3i_system->user_interface
manage_user_interface's ancestor chain: c3i_system->user_interface
message_arrival_panel's ancestor chain: c3i_system->user_interface
message_editor's ancestor chain: c3i_system->user_interface
status_screen's ancestor chain: c3i_system->user_interface
weapons_interface's ancestor chain: c3i_system
weapons_systems's ancestor chain: c3i_system

Test-Case: test_out_file

This test-case successfully demonstrated get and put of c3i_system.NEW.psdl. This file was created from the prototype reconstructed in test_dg_2. The test driver and output are not listed for this test-case.

LIST OF REFERENCES

[1] Berzins, V., and Dampier, D., "Software Merge: Combining Changes to Decompositions", *Journal of Systems Integration*, Vol. 6, Num. 2, Kluwer, 1996, pp. 135-150.

[2] Dampier, D., "A Formal Method for Semantics-Based Change-Merging of Software Prototypes", *Ph.D. Dissertation*, Naval Postgraduate School, Monterey, California, June 1994.

[3] Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, pp. 1409-1423, October 1988.

[4] Dampier, D., Luqi, and Berzins, V., "Automated Merging of Software Prototypes", *Journal of Systems Integration*, Vol. 4, Num. 1, Kluwer, February 1994, pp. 33-49.

BIBLIOGRAPHY

Berzins, V., *Software Merging and Slicing*, IEEE Computer Society Press, 1995.

Berzins, V., and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, Reading, Mass., 1991.

Cohen, N. H., *Ada as a Second Language*, McGraw-Hill, 1996

Proc. ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer Aided Software Development: Software Slicing, Merging, and Integration, U. S. Naval Postgraduate School, Monterey, Calif., 1993.

INITIAL DISTRIBUTION LIST

No. of copies

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   411 Dyer Rd.
   Naval Postgraduate School
   Monterey, CA 93943

3. Center for Naval Analysis 1
   4401 Ford Ave.
   Alexandria, VA 22302

4. Dr. Ted Lewis, Chairman, Code CS/LT.. 1
   Computer Science Dept.
   Naval Postgraduate School
   Monterey, CA 93943

5. Chief of Naval Research 1
   800 North Quincy St.
   Arlington, VA 22217

6. Dr. Luqi, Code CS/Lq 1
   Computer Science Dept.
   Naval Postgraduate School
   Monterey, CA 93943

7. Dr. Marvin Langston 1
   1225 Jefferson Davis Highway
   Crystal Gateway 2 / Suite 1500
   Arlington,VA 22202-4311

8. David Hislop 1
   U.S. Army Research Office
   PO Box 12211
   Research Triangle Park, NC 27709-2211

9. Capt. Talbot Manvel 1
Naval Sea Systems Command
2531 Jefferson Davis Hwy.
Attn: TMS 378 Capt. Manvel
Arlington,VA 22240-5150

10. CDR Michael McMahon 1
Naval Sea Systems Command
2531 Jefferson Davis Hwy.
Arlington, VA 22242-5160

11. Dr. Elizabeth Wald 1
Office of Naval Research
800 N. Quincy St.
ONR CODE 311
Arlington, VA 22132-5660

12. Dr. Ralph Wachter 1
Office of Naval Research
800 N. Quincy St.
CODE 311
Arlington, VA 22217-5660

13. Army Research Lab 1
115 O'Keefe Building
Attn: Mark Kendall
Atlanta, GA 30332-0862

14. National Science Foundation 1
Attn: Bruce Barnes
Div. Computer & Computation Research
1800 G St. NW
Washington, DC 20550

15. National Science Foundation 1
Attn: Bill Agresty
4201 Wilson Blvd.
Arlington, VA 22230

16. Hon. John W. Douglas 1
Assistant Secretary of the Navy
(Research, Development and Acquisition)
Room E741
1000 Navy Pentagon
Washington, DC 20350-1000

17. Commanding Officer 1
Attn: William R. Keesling
Naval Command, Control and Ocean Surveillance Center
RDT&E DIV D7213
53150 Systems Street Rm. 109
San Diego, CA 92152-9512

18. Technical Library Branch 1
Naval Command, Control and Ocean Surveillance Center
RDT&E DIV D0724
San Diego, CA 92152-5001